Proyecto Fin de Carrera

Facultad de Informática
Universidad de Murcia
Murcia, Septiembre de 2008

Virtual-GEMS

Una infraestructura de simulación para máquinas virtuales

Alumno:
Antonio García Guirado

Directores:
Ricardo Fernández Pascual
José Manuel García Carrasco

Resumen

En este trabajo se expone la importancia del uso de simuladores en el diseño de computadores, así como la importancia de la virtualización para la consolidación de servidores. Se desarrolla una infraestructura de virtualización que permite realizar la simulación de varias máquinas virtuales corriendo sobre un mismo hardware, permitiéndonos probar el impacto en el rendimiento que producen los distintos parámetros del computador configurables en la infraestructura. Finalmente, se estudia la aplicación de nuestra infraestructura a algunos casos concretos.

Índice general

1. Introducción **9**

1.1. Objetivos 10

2. Fundamentos y Estado del Arte **13**

2.1. Simulación en arquitectura de computadores 13

2.2. Simuladores, una taxonomía 14

2.2.1. Por tipo de simulación 14

2.2.2. Por método de simulación 15

2.2.3. Por intervalo en la simulación 16

2.2.4. Por los sistemas soportados 16

2.2.5. Por nivel de detalle en la simulación 16

2.2.6. Simuladores modulares 17

2.3. Algunos de los simuladores más utilizados 17

2.3.1. SimpleScalar 17

2.3.2. Simics 18

2.3.3. RSIM 18

2.3.4. Asim (A performance model framework) 19

2.3.5. GEMS 19

2.4. Introducción a la virtualización 21

2.5. Conceptos básicos de virtualización 23

2.6. Maquinas virtuales de proceso 24

2.6.1. Multiprogramación 24

2.6.2. Emuladores y traductores binarios dinámicos 24

2.6.3. Optimizadores de binarios de mismo ISA 25

2.6.4. Maquinas virtuales de lenguajes de alto nivel. Independencia de la plataforma 25

2.7. Máquinas virtuales de sistema 25

2.7.1. Emulación. Máquinas virtuales de sistema completo 26

2.7.2. Codesigned virtual machines 26

2.8. Razones para virtualizar 27

2.9. Relación entre simuladores y virtualización 28

3. Simics: elementos y programación del simulador 29
3.1. Módulos en Simics 29
3.1.1. Clases y objetos 30
3.1.2. Interfaces 30
3.1.3. Eventos 30
3.1.4. Haps 31
3.1.5. Comandos 31
3.2. Interfaz de Jerarquía de Memoria 31
3.2.1. Deteniendo las transacciones de memoria 33
3.3. Pasos y ciclos en Simics 34
3.4. Un módulo que implementa la interfaz timing-model 34

4. GEMS 37
4.1. Driver de Ruby 37
4.2. Drivers existentes en GEMS 37
4.3. Ruby 39
4.3.1. Componentes independientes del protocolo 39
4.3.2. Funcionamiento de Ruby 39

5. Desacoplamiento de Simics y GEMS 45
5.1. Elementos de la solución 45
5.2. Modulo mi-device 45
5.3. Manejo de las peticiones de memoria 46
5.4. Módulo RemoteRuby 47
5.5. Comunicación entre mi-device y RemoteRuby 49
5.5.1. Comunicación adicional 51
5.6. Comandos de mi-device 52

6. Virtualización con Virtual-GEMS 53
6.1. Primeras consideraciones 53
6.2. Maquinas virtuales de sistema 55
6.3. Máquinas virtuales nativas y hosted 56
6.4. Virtualización de recursos 56
6.4.1. Procesador 57
6.4.2. Memoria 58
6.4.3. Entrada/salida 61
6.5. Virtualización en multiprocesadores 61

6.5.1. Mecanismos para soportar particionado 62
6.5.2. Particionado físico 62
6.5.3. Particionado lógico 62
6.6. Nuestra implementación 63
6.7. Estructura de Virtual-GEMS 65
6.8. Comunicación 65
6.8.1. Convención sobre triggerEvents 67
6.9. Estadísticas 68
6.10. Cargas usadas 68
6.10.1. Preparación de las cargas 69
6.10.2. Problemas con las cargas 70

7. Validación de Virtual-GEMS 73
7.1. Arquitecturas simuladas 73
7.2. Máquinas virtuales 75
7.3. Mapeo de los procesadores 76
7.4. Mapeos de memoria 77
7.5. Protocolos de coherencia 82
7.6. Resultados 85

8. Conclusiones y vías futuras 91

A. Uso de Virtual-GEMS 93

Índice de figuras

2.1. Definición formal de la virtualización 22
2.2. Interfaz de máquinas 23

3.1. Espacio de memoria en Simics 32
3.2. Interfaz timing-model 35

4.1. Esquema de Ruby 41
4.2. Sincronización de Simics y Ruby 42
4.3. Estructura del módulo SimicsRuby 43

5.1. División de SimicsRuby en mi-device y RemoteRuby 46

6.1. Virtualización en el simulador 54
6.2. Mapeo de memoria. Tabla de páginas de dos niveles. 59
6.3. Nuestro esquema de virtualización 64
6.4. Esquema de la virtualización con Virtual-GEMS 66
6.5. Fases de ejecución de un *workload* 70
6.6. Fases de ejecución de los *workloads* virtualizados 70
6.7. *Workloads* con transacciones extra durante la espera 71

7.1. Estructura del Tiled CMP 74
7.2. Mapeo de la memoria en los *slices* de L2 de un chip 74
7.3. Estructura del M-CMP 75
7.4. Mapeo de los procesadores de las MVs en el *tiled* CMP 76
7.5. Mapeo de los procesadores de las MVs en el M-CMP 77
7.6. Mapeo de memoria real a física simple 78
7.7. Rangos de bits en una dirección de memoria 79
7.8. Mapeo de memoria real a física VMCBM 80
7.9. Diferencia entre *tiled* CMP con mapeo VMCBM y M-CMP con mapeo simple 81
7.10. Resultados del protocolo DIRECTORY 87
7.11. Resultados del protocolo TOKEN 87
7.12. Speedup con respecto al caso base con DIRECTORY 88

7.13. Speedup con respecto al caso base con TOKEN 88
7.14. Mejora de TOKEN con respecto a DIRECTORY 89

1

Introducción

Para gran parte de la investigación y diseño en arquitectura de computadores, la evaluación cuantitativa de las arquitecturas futuras es sólo posible mediante el uso de simuladores. Los simuladores reducen el coste y el tiempo de un proyecto, permitiendo al arquitecto evaluar el rendimiento de un amplio rango de arquitecturas sin necesidad de construirlas.

Desde principios de los 80, el diseño de computadores de alto rendimiento ha usado generalmente una aproximación dirigida por los datos. Por ejemplo, el análisis del uso de instrucciones reveló que las máquinas reales no usaban todas las instrucciones con la misma frecuencia. Los diseñadores usaron esta observación para optimizar la implementación de estas máquinas.

La toma de medidas de rendimiento directas, sin embargo, es un paso posterior a la hora de realizar el diseño, ya que necesitamos tener construido una parte del simulador o la máquina real de la que podamos tomar dichas medidas, y no siempre ayuda a optimizar las máquinas que se están diseñando. Como alternativa, los diseñadores adoptaron modelos analíticos para predecir el rendimiento. Estos modelos tenían éxito en muchos casos, particularmente en seleccionar el espacio de diseño en exploraciones preliminares. Sin embargo, los modelos analíticos han tenido menos éxito al evaluar propuestas específicas de un diseño determinado.

Por su parte, la virtualización se ha convertido en una técnica ampliamente usada en diversos ámbitos. La virtualización nos permite aprovechar los recursos físicos de una máquina, de manera que dé la impresión de que disponemos de una o varias máquinas virtuales, que pueden ser iguales o distintas a la máquina física. Gracias a ello podemos ejecutar varios SOs sobre una misma máquina física, creyendo cada uno de ellos que tiene una máquina completa a su disposición; o también podemos ejecutar software compilado para una arquitectura en una máquina de una arquitectura distinta, encargándose el sistema de virtualización de hacer la traducción entre ambas arquitecturas.

Uno de los usos principales de la virtualización está en la consolidación de servidores. Es frecuente que en un centro de datos dispongamos de varios servidores, ejecutando cada uno de ellos aplicaciones diversas, como servicios web o gestores de bases de datos. La consolidación de servidores consiste en llevarnos todos estos servicios a un único servidor físico. Gracias a la virtualización esto es posible, de manera que el software que se ejecutaba en ca-

da servidor sigue teniendo la impresión de que se ejecuta en una máquina propia, aunque en realidad correrá sobre una máquina virtual. Así se simplifica la gestión y mantenimiento del centro de datos, ya que ahora puede constar hasta de un único servidor. Además, el sistema de virtualización hará que los recursos del servidor se aprovechen mejor, repartiéndolos y balanceándolos entre las distintas máquinas virtuales según los requisitos de potencia de cómputo que éstas tengan en cada momento. Así conseguimos además que la capacidad de procesamiento total necesaria sea menor que teniendo varios servidores independientes.

A pesar de la importancia de la virtualización, existe una carencia de herramientas que nos permitan la investigación en este área. Los simuladores que suelen usarse para investigar nuevas arquitecturas no proporcionan mecanismos para poder simular directamente máquinas virtuales, ni son lo suficientemente detallados y precisos como para poder ejecutar sobre ellos el software de virtualización. Parece por tanto claro que son necesarios nuevos instrumentos que permitan investigar en virtualización de la misma manera que se hace en otras áreas de arquitectura de computadores, es decir, mediante simuladores.

Nuestro propósito principal en este proyecto es el de desarrollar una infraestructura de simulación para máquinas virtuales que proporcione la capacidad de simular máquinas virtuales que se ejecutan sobre un mismo sistema físico. Partiendo del simulador Simics (que fue ideado permitiendo que fuera expandido de una manera relativamente sencilla), y de un módulo que a su vez expande a Simics, llamado GEMS (que es el encargado de simular el rendimiento o temporización del sistema de memoria de la máquina simulada), construiremos nuestra infraestructura.

1.1. Objetivos

Este proyecto tiene varios objetivos. En primer lugar, proporcionar una infraestructura de simulación de máquinas virtuales. En segundo, mostrar, usando la infraestructura recién creada, como con pequeños cambios en el sistema de memoria, tales como modificaciones del protocolo de coherencia o cambios en el mapeo de los elementos de las máquinas virtuales sobre el hardware, se obtienen mejoras de rendimiento nada despreciables con respecto a una máquina convencional sobre la que se ejecutase un sistema de virtualización.

Además, uno de los pasos que daremos para construir la infraestructura será desacoplar el simulador de temporización de sistemas de memoria GEMS y el simulador de sistemas completos Simics. GEMS es un módulo de Simics que proporciona a éste la capacidad de calcular el rendimiento del sistema completo simulado realizando la simulación de temporización de un sistema de memoria configurable. Tras desacoplar ambos simuladores, podremos usar GEMS para simular un sistema de memoria dentro de cualquier otro simulador de máquinas que permita ser expandido, con tan sólo construir un pequeño módulo que se encargue de pasar a GEMS las peticiones de memoria, siguiendo una interfaz simple y bien definida de comunicación con GEMS. Además, también podrá usarse la infraestructura de virtualización con cualquier otro simulador al que conectemos GEMS (es decir, no sólo con Simics).

En este documento se muestran los pasos seguidos para conseguir todos los objetivos que acabamos de exponer. A continuación mostramos, de manera breve, qué podemos encontrar

en estas páginas. En el Capítulo 2 damos una visión sobre los dos temas alrededor de los cuales gira este proyecto: simulación y virtualización. Veremos los usos de la simulación, una taxonomía de los simuladores existentes, y las características de algunos de los simuladores más usados en arquitectura, entre ellos los dos simuladores en los que se basa nuestro trabajo: Simics y GEMS. En cuanto a la virtualización, veremos los conceptos básicos para comprender su funcionamiento.

En el Capítulo 3 abordamos en detalle el simulador Simics y creamos para éste un primer módulo que modela el rendimiento del sistema de memoria.

El Capítulo 4 explica el funcionamiento del simulador de sistemas de memoria GEMS, y el Capítulo 5 trata el desacoplado de Simics y GEMS para conseguir independizar el sistema de memoria (GEMS) de un simulador concreto (Simics), en lo que supone un paso intermedio para la construcción de nuestra infraestructura. El Capítulo 6 está dedicado a la construcción de nuestro sistema de virtualización. Una vez desacoplados GEMS y Simics veremos como crear un sistema virtualizado usando varios procesos de Simics como si de máquinas virtuales se tratase, necesitando una única instancia de GEMS para simular el sistema de memoria físico. También veremos los problemas que surgen al usar simultáneamente simulación, virtualización y cargas de trabajo prefabricadas.

El Capítulo 7 está dedicado a la validación de nuestra infraestructura Virtual-GEMS. Primero describimos las arquitecturas que usaremos en las pruebas, tras lo que mostraremos los resultados de la ejecución de un conjunto de benchmarks y las mejoras de rendimiento obtenidas dependiendo de los parámetros utilizados.

Finalmente, en el Capítulo 8 se comentan las posibles vías de investigación que se abren una vez que disponemos de la infraestructura de simulación de máquinas virtuales Virtual-GEMS. Se acompaña un Anexo A que explica como usar el software desarrollado.

2

Fundamentos y Estado del Arte

2.1. Simulación en arquitectura de computadores

Cualquier ordenador puede simular a cualquier otro. Así lo demuestra el trabajo teórico de Alan Turing y Alonzo Church. Esta propiedad se aprovechó ya en los años 50, dentro del proyecto EDSAC [11], y desde entonces la simulación en sus distintas formas se ha usado para facilitar el diseño de computadores.

Los simuladores proporcionan gran flexibilidad para modificar y analizar el impacto de los diversos parámetros arquitecturales y componentes, así como permiten una recogida de estadísticas mucho más detalladas que usando hardware real. Además, una de las principales ventajas de los simuladores es evitar tener que construir máquinas reales para probar diversos parámetros de diseño, con el gran coste que ello supondría.

Los simuladores, sin embargo, están limitados por dos problemas. El primero es la velocidad de la simulación. Debido a que los simuladores normalmente simulan programas que teóricamente deben ser ejecutados directamente sobre el hardware, y a que la velocidad de los simuladores precisos es varios órdenes de magnitud más lenta que la ejecución sobre el hardware real, simular exactamente los mismos programas y con las mismas entradas que se piensan ejecutar en el hardware ya desarrollado es virtualmente imposible. Por ello, para minimizar el tiempo de ejecución se suele usar tan sólo un subconjunto de las aplicaciones objetivo, usar conjuntos de entradas reducidos, o simular un conjunto representativo de intervalos de la ejecución de cada programa.

El segundo problema es la posible falta de precisión. Este problema es especialmente nocivo ya que prácticamente cada decisión que tomen tanto el desarrollador del software de simulación como el investigador que hace uso de dicho software, a la hora de realizar la simulación, puede degradar la exactitud de la misma, y la cantidad de error es difícil de determinar a posteriori.

La simulación ofrece los beneficios tradicionales de la construcción del software: permite parametrizar, controlar e inspeccionar el sistema que se modela (el sistema objetivo). Las medidas obtenidas son deterministas y no intrusivas. Además, la simulación proporciona

las bases para la automatización: múltiples sesiones de simulación pueden ejecutarse en paralelo, y dichas sesiones pueden estar totalmente automatizadas.

Los modelos software pueden usarse para desarrollar y probar software antes de que el hardware real sobre el que se ejecutará esté disponible. Aunque los modelos software son más lentos que su contrapartida en hardware, el programador puede construirlos y probarlos en menos tiempo y a un coste mucho menor que el necesario para construir hardware real. Este rápido mecanismo para diseñar y probar proporciona un tiempo menor hasta llevar el producto al mercado y una producción inicial del hardware de mayor calidad.

Tres aspectos críticos dirigen la implementación de un modelo software o simulador: *rendimiento, flexibilidad y detalle*. El rendimiento determina la cantidad de carga de trabajo que el modelo puede soportar dados los recursos disponibles para realizar la simulación. La flexibilidad indica cómo de bien está estructurado el modelo para facilitar su modificación, permitiendo modelar con facilidad variantes del diseño o incluso diseños totalmente diferentes. El detalle define el nivel de abstracción usado al implementar los componentes del modelo. Un modelo con gran detalle simulará fielmente todos los aspectos del funcionamiento de la máquina, independientemente de si un aspecto en particular es importante para la magnitud que se esté midiendo.

En la práctica es difícil optimizar estas tres características de manera simultanea. Por ello, la mayoría de implementaciones de modelos software optimizan sólo una o dos de ellas, lo que explica por qué existen tantos simuladores distintos, incluso para un mismo diseño de máquina. Los modelos de investigación suelen optimizar el rendimiento y la flexibilidad a expensas del detalle.

Los simuladores de computadores pueden clasificarse en varios tipos. Por un lado estarían aquellos que simulan el rendimiento de un monoprocesador, como SimpleScalar [3], frente a simuladores que pueden simular máquinas paralelas, como RSIM [13]. Por otra parte tenemos simuladores de sistemas completos, como Simics [14]. También existen simuladores de consumo de potencia, como Wattch [5]. Por último tendríamos los simuladores modulares, como Liberty [26]. En el siguiente apartado veremos una división más formal de los simuladores según su propósito y la forma de implementarlos.

2.2. Simuladores, una taxonomía

Existen varias clasificaciones de los simuladores de computadores. Yi et al. [31] muestran algunas taxonomías basándose en diversos criterios. A continuación presentamos nuestra propia taxonomía atendiendo a los aspectos más interesantes de los simuladores con respecto a sus posibles usos en la investigación en el área de ingeniería de computadores.

2.2.1. Por tipo de simulación

Normalmente, hay dos componentes típicos en una simulación:

Simulación funcional: Necesaria para comprobar la corrección. Simula el comportamiento de la máquina real y modela dispositivos comunes como discos o interfaces de vídeo o de red.

Simulación de temporización: Usada para medir el rendimiento. Modela la latencia de operación de los dispositivos emulados por el simulador funcional y se asegura de que los eventos generados por esos dispositivos se simulan en un orden temporal correcto.

Más recientemente, la **simulación de consumo energético** ha cobrado importancia, especialmente al analizar los costes energéticos a nivel de centro de datos. Al igual que con la simulación de temporización, la simulación funcional debe proporcionar eventos de la CPU y dispositivos a los que podamos aplicar el modelo energético para estimar el consumo global del sistema.

2.2.2. Por método de simulación

Según esta clasificación distinguimos los siguientes tipos de simuladores:

- Los **simuladores dirigidos por trazas** disminuyen el tiempo de simulación reduciendo la sobrecarga de la simulación funcional. Emplean un simulador funcional para simular la aplicación objeto una vez, guardan una traza de los eventos interesantes, y usan varias veces la traza de eventos almacenada con diferentes modelos de temporización para estimar el rendimiento (o consumo energético, si es lo que deseamos medir). Una limitación grave de este tipo de simulador es la imposibilidad de proporcionar retroalimentación dependiente del tiempo al comportamiento de la aplicación. Por esta razón, los enfoques dirigidos por trazas que funcionan correctamente para aplicaciones monohilo ejecutadas en un monoprocesador son menos apropiados para la simulación de sistemas completos, especialmente si se trata de sistemas paralelos. Por ejemplo, muchas librerías multihilo usan bucles de espera activa por sus ventajas en esperas cortas. Los protocolos de red pueden tener que reenviar paquetes dependiendo de la congestión del sistema. En este y muchos otros escenarios la retroalimentación temporal es fundamental para una simulación precisa del sistema. Sobre todo, cobra importancia en el orden de adquisición de candados por parte de distintos procesadores que compiten.

- Los **simuladores basados en la ejecución de programas** (*execution-driven*) realizan directamente la ejecución del simulador funcional junto con el modelo de temporización. Sin embargo, estos simuladores suelen ser varios órdenes de magnitud más lentos que el hardware nativo, debido a la sobrecarga causada al aplicar la simulación de temporización para cada instrucción emulada. Por ejemplo, consideremos SimpleScalar, un simulador de arquitecturas dirigido por la ejecución de programas usado comúnmente. La velocidad típica de ejecución del simulador en un ordenador capaz de ejecutar 1000-2000 MIPS es de 6 o 7 millones de instrucciones simuladas por segundo [9]. Esto significa una disminución de dos a tres órdenes de magnitud. Si añadimos simulación de temporización, la velocidad cae a aproximadamente $0,3$ MIPS, es decir, otros 1 o 2 órdenes de magnitud. Por tanto, la simulación puede fácilmente ser 10000 veces más lenta que la ejecución nativa. En la práctica, esta sobrecarga restringe la

aplicabilidad de la simulación dirigida por la ejecución de programas a escenarios de aplicaciones monohilo ejecutadas en máquinas monoprocesador.

2.2.3. Por intervalo en la simulación

Tenemos simuladores dirigidos por tiempo (o ciclo a ciclo) y simuladores dirigidos por eventos. Un simulador dirigido por tiempo avanza por incrementos de tiempo fijos, normalmente ciclo a ciclo, y es útil para modelar eventos que ocurren a intervalos de tiempo regulares. Un simulador dirigido por eventos avanza incrementos de tiempo irregulares y es útil para modelar eventos que pueden ocurrir en cualquier instante. Un simulador dirigido por eventos es más eficiente que uno dirigido por tiempo ya que no avanza a intervalos de tiempo regulares cuando no ocurren eventos.

2.2.4. Por los sistemas soportados

Cada simulador es capaz de modelar un conjunto de sistemas. Entre ellos pueden encontrarse sistemas paralelos, sistemas monoprocesador, sistemas móviles o sistemas embebidos, dentro de una amplia variedad.

En nuestro caso, la diferenciación más importante está entre los simuladores que modelan un sistema simple con un único procesador, como SimpleScalar, y aquellos que permiten ejecutar simulaciones con sistemas multiprocesador, como RSIM.

2.2.5. Por nivel de detalle en la simulación

Según esta característica podemos dividir los simuladores en dos tipos:

Simuladores de proceso o aplicación: Simulan la ejecución directa de una aplicación sobre la máquina. Suelen proporcionar un conjunto de herramientas (como compiladores y ensambladores) para poder generar nuestras propias aplicaciones que usar en el simulador.

Simuladores de sistema completo: Simulan la máquina completa con tal exactitud que permiten la ejecución de cualquier software sobre ella. Necesitan un nivel de detalle mucho mayor para poder así ejecutar un sistema operativo, y sobre él, cualquier software que precisemos simular.

Que un simulador sea de proceso no significa que el tipo de máquina a simular esté limitado: puede modelar máquinas multiprocesador perfectamente, como ocurre con el simulador RSIM. Esta diferenciación entre simulación de aplicación y simulación de sistema completo es muy importante. El no emular el sistema operativo puede causar una gran variación de los resultados obtenidos. Cain et al. [6] demostraron que omitir el SO puede producir errores de hasta un 100 % en los resultados de la simulación de los benchmarks SPEC CPU 2000.

2.2.6. Simuladores modulares

Una característica muy importante hoy día en los simuladores es la modularidad del simulador. Ejemplos de simuladores modulares pueden ser Liberty [27] o Simics, que permiten agregar al sistema simulado elementos en forma de módulos, que nosotros mismos podemos desarrollar.

Conforme aumenta la complejidad de las máquinas simuladas, la implementación y depuración del simulador pueden ser muy costosas en tiempo, su precisión puede verse reducida debido a un detalle insuficiente, y la reutilización de componentes es complicada.

Para minimizar los efectos de estos problemas, los arquitectos han propuesto el uso de simuladores modulares como el Entorno de Simulación Liberty (LSE). Desarrollado en Princeton, Liberty mapea cada componente del hardware a un único elemento en el software. Instanciando estos componentes y especificando sus conexiones, los arquitectos pueden construir jerárquicamente componentes más complejos. Esta construcción jerárquica tiene dos ventajas clave. Primero, realizar la construcción y prueba de los componentes de más bajo nivel de manera jerárquica permite a los arquitectos construir fácilmente grandes bibliotecas de componentes precisos. Segundo, el arquitecto puede construir rápida y fácilmente un amplio rango de modelos, lo que le permite explorar eficientemente el espacio de diseño y examinar arquitecturas más inusuales. Para asegurarse de que los componentes futuros interaccionen con los actuales, LSE estipula la interfaz de control entre los componentes LSE. Por ejemplo, se especifican las señales de “enable” y “ack” que deben usarse para sincronizar la transferencia de datos entre los componentes.

Como desventaja tenemos que estos simuladores suelen ser especialmente lentos, debido a la sobrecarga de procesamiento que añaden las interacciones que deben realizarse entre los numerosos y pequeños módulos que conforman el sistema simulado.

2.3. Algunos de los simuladores más utilizados

2.3.1. SimpleScalar

El simulador SimpleScalar [3] fue escrito en 1992 y publicado en 1995. Se ha convertido en uno de los simuladores más populares. Consiste en una serie de simuladores y en las herramientas necesarias para darles soporte (como un compilador gcc, sus librerías, un ensamblador, y un enlazador o linker). Los diferentes simuladores que se proporcionan cubren un amplio rango de compromiso entre velocidad y detalle. Puede modelar una gran variedad de plataformas, desde simples procesadores no segmentados hasta microarquitecturas dinámicamente planificadas con jerarquías de memoria de múltiples niveles. Una limitación que presenta es que sólo simula arquitecturas monoprocesador.

El diseño de SimpleScalar está bien estructurado y documentado, lo que simplifica la tarea de extenderlo para abordar tareas muy variadas de modelado de arquitecturas.

2.3.2. Simics

Aunque simuladores como SimpleScalar modelan la microarquitectura, no modelan el sistema completo ni ejecutan un sistema operativo (SO). En la sección 2.2.5 vimos que omitir el SO puede producir errores tan altos como del 100 % en la simulación. Para evitar estos problemas, los arquitectos usan simuladores de sistemas completos que modelan el sistema entero con la suficiente profundidad como para ejecutar el SO.

Sería por tanto deseable poder simular un sistema completo y hacerlo con total precisión (obtener un modelo perfecto). Hay problemas obvios en buscar esta perfección, incluyendo coste, tiempo, inexactitudes en la especificación y errores en la implementación. Pero aún más importante es el problema del realismo de las cargas de trabajo (o *workloads*). En la mayoría de casos no es posible implementar un modelo preciso con el suficiente rendimiento como para ejecutar cargas realistas. Por ello, en la práctica, los modelos que intentan ser exactos terminan ejecutando cargas de juguete demasiado pequeñas. El resultado es que obtenemos respuestas muy precisas a preguntas irrelevantes.

Simics [14] es una plataforma para la simulación de sistemas completos, que intenta alcanzar un equilibrio entre precisión y rendimiento. Es decir, es lo suficientemente abstracto para alcanzar un nivel de rendimiento tolerable y, al mismo tiempo, tiene la suficiente precisión *funcional* para ejecutar cargas comerciales y la suficiente precisión en la *temporización* como para interactuar con modelos hardware detallados.

Simics, que apareció en el año 2002, fue uno de los primeros proyectos académicos en este área. Además fue el primer simulador de sistemas completos comercial y ayudó a demostrar las posibilidades de este tipo de simulación en el desarrollo de sistemas. Su predecesor es SimOS [23], un simulador de sistemas completos creado en la Universidad de Stanford.

Simics soporta un amplio rango de conjuntos de instrucciones, incluyendo Alpha, PowerPC, UltraSparc y x86. Además incluye modelos de dispositivos, como tarjetas gráficas, tarjetas de red ethernet, o controladores de bus. Estos modelos son tan detallados que permiten ejecutar los *drivers* reales de los dispositivos. Además, Simics puede ejecutar sistemas operativos sin necesidad de modificarlos, tales como Linux, Solaris o Windows XP. Entre las características más interesantes de Simics podemos encontrar la capacidad de establecer *breakpoints* y continuar luego la simulación, realizar y cargar *checkpoints* (instantáneas del estado de la máquina) y usar el llamado "Simics Central" que nos permite simular varias máquinas distintas de manera sincronizada.

2.3.3. RSIM

RSIM (Rice Simulator for ILP Multiprocessors) [13] fue desarrollado en Rice y publicado en 1997. Sus características principales son el lanzamiento fuera de orden de instrucciones, renombrado de registros, predicción de saltos, cachés no bloqueantes, múltiples protocolos de coherencia de caché y parámetros configurables por el usuario como el tamaño de la ventana de instrucciones, el tamaño y latencia de las cachés, el retardo de la red o su ancho de banda. Fue desarrollado para correr en máquinas SPARC ejecutando el sistema operativo Solaris. Ha sido portado a GNU/Linux corriendo sobre arquitectura x86 por el Grupo de investigación GACOP de la Universidad de Murcia [10].

2.3.4. Asim (A performance model framework)

Los microprocesadores hoy en día han superado la barrera de los mil millones de transistores. La interacción entre las estructuras hardware construidas con esos millones de transistores es difícil de predecir mediante modelos de simulación o análisis. Tener tal número de transistores ha hecho que los microprocesadores modernos sean extremadamente complejos, alargando el ciclo de diseño de nuevas microarquitecturas. Aún es más, debido a que los aspectos arquitecturales del diseño se ven afectados cada vez más por los retardos de propagación de las señales, modelar la temporización de estos retardos de forma sistemática se ha hecho más importante. En este contexto, los modelos de rendimiento juegan un papel crucial porque los desarrolladores los usan para estudiar alternativas de diseño y predecir el rendimiento de los procesadores y sistemas antes de construirlos realmente.

Estos factores han resultado en un incremento en la complejidad de los modelos de rendimiento en sí mismos. En el pasado, la longevidad y utilidad de un modelo dependían principalmente en las habilidades y disciplina de los escritores del mismo. Desafortunadamente, a pesar de esta habilidad, los modelos se han vuelto extremadamente complejos e inmanejables debido a la falta de una manera estructurada de desarrollarlos.

Una aproximación a este problema es la tomada en Compaq (anteriormente Digital, y actualmente propiedad de HP). A finales de 1998 comenzaron a desarrollar Asim [8], que permite a los escritores de modelos representar la temporización detallada de las complejas máquinas modernas y manejar de manera efectiva los grandes proyectos software necesarios para modelar estas máquinas.

Asim aborda estas necesidades proporcionando una infraestructura para crear muchos modelos, en lugar de ser en sí mismo un único modelo de rendimiento. Más específicamente, Asim consigue estos objetivos a través de la modularidad y reusabilidad. El uso de módulos ayuda a separar el problema del modelado de rendimiento en partes individuales que pueden ser modeladas de manera independiente, mientras que la reusabilidad aumenta la productividad y la confianza en la robustez del componente software en sí, ya que podemos aprovechar los módulos existentes sabiendo que son correctos.

2.3.5. GEMS

El Proyecto Multifacet de Wisconsin ha creado un conjunto de herramientas de simulación para caracterizar y evaluar el rendimiento de sistemas multiprocesadores usados comúnmente como servidores web y de bases de datos. Basándose en la infraestructura de simulación funcional de sistemas completos proporcionada por Simics, se construye un conjunto de módulos de simulación de temporización para modelar el rendimiento del sistema de memoria y los microprocesadores. Esta infraestructura de simulación nos permite ejecutar experimentos arquitecturales usando un juego de cargas comerciales reducidas. Estos módulos de simulación de temporización forman Multifacet GEMS (General Execution-driven Multiprocessor Simulator).

Como hemos comentado, la simulación es una de las técnicas más importantes usada por los arquitectos de computadores para evaluar sus innovaciones. Una de las cuestiones

más importantes es que no sólo es necesaria una simulación detallada de la máquina objeto, sino que debemos usar una carga de trabajo realista.

Por ejemplo, SimpleScalar se ha usado ampliamente en investigación para evaluar nuevas ideas. Sin embargo, éste y otros simuladores sólo ejecutan trabajo en modo usuario y con un sólo hilo, como por ejemplo los benchmarks del SPEC CPU. Muchos diseñadores están interesados en sistemas multiprocesadores que pueden ejecutar cargas más complicadas, con varios hilos, tales como bases de datos, servidores web o aplicaciones científicas paralelas. Estas cargas dependen de muchos servicios del sistema operativo (por ejemplo entrada/salida, sincronización, planificación y migración). Además, conforme los microprocesadores de un único chip han evolucionado a arquitecturas multinucleo, se ha hecho más necesaria la capacidad de simular estas máquinas ejecutando cargas multihilo realistas, para poder así continuar con la investigación en arquitectura.

Crear un simulador de temporización o rendimiento para evaluar sistemas multiprocesador con cargas que requieren soporte del sistema operativo es difícil. Primero, incluso crear un simulador funcional, que no proporciona modelado del tiempo, es una tarea considerable. Proporcionar la suficiente fidelidad funcional para arrancar un sistema operativo sin modificar requiere implementar las instrucciones en modo supervisor e interactuar con modelos funcionales de muchos dispositivos de entrada/salida. Tales simuladores son aquellos a los que llamábamos simuladores de sistemas completos. Segundo, crear un simulador de temporización detallado que ejecute sólo código a nivel de usuario es también complicado. Por último, crear un conjunto de herramientas que soporte tanto simulación funcional del sistema completo como simulación de temporización es una tarea sustancialmente más complicada que cualquiera de ellas por separado.

GEMS [17] es una infraestructura de simulación modular que desacopla la simulación de la funcionalidad de la de la temporización. Se usa para ello una aproximación “timing-first”. Los módulos de temporización interactúan con Simics indicándole cuando debe ejecutar una instrucción. Sin embargo, el resultado de la ejecución de la instrucción depende de Simics. Esta aproximación es distinta a la de las simulaciones dirigidas por trazas. Aunque en ambos casos la simulación funcional y la simulación de temporización se encuentran desacopladas, usando la aproximación “timing-first” el simulador funcional se ve afectado por el simulador de temporización, permitiendo al sistema capturar los efectos dependientes de la temporización (o del rendimiento). Por ejemplo, el modelo de temporización determinará el ganador de dos procesadores que están intentando acceder al mismo cerrojo en memoria. Ya que el simulador de temporización determina cuando avanza el simulador funcional, esos efectos dependientes del tiempo son capturados. Por el contrario, una simulación dirigida por trazas no puede capturar estos importantes efectos.

GEMS fue construido para estudiar sistemas de memoria con coherencia de caché y temas relacionados, por lo que estos aspectos son los más detallados y flexibles de la infraestructura de simulación. Por ejemplo, se modelan los estados transitorios de los protocolos de coherencia en gran detalle. Sin embargo, el modelado de la red de interconexión, el subsistema de DRAM y la entrada/salida no es tan detallado, si bien las últimas versiones del simulador han mejorado mucho estos aspectos.

2.4. Introducción a la virtualización

La clave para manejar la complejidad en los computadores está en su división en niveles de abstracción separados por interfaces bien definidas. Los niveles de abstracción permiten que los detalles de implementación de los niveles más bajos del diseño sean ignorados o simplificados, facilitando por tanto el diseño de los componentes en niveles más altos.

Los niveles de abstracción se organizan en una jerarquía, con los niveles más bajos implementados en hardware y los más altos en software. Al hablar de virtualización estamos interesados en los niveles cercanos a la frontera hardware/software.

El software se ejecuta en una **máquina**. Desde la perspectiva del sistema operativo, una máquina está principalmente compuesta por recursos, incluyendo uno o más procesadores que utilizan un conjunto de instrucciones específico (ISA), alguna memoria real, y dispositivos de E/S. Sin embargo el término máquina no se limita al hardware. Desde la perspectiva de una aplicación, la máquina es una combinación de sistema operativo y las porciones del hardware accesibles mediante instrucciones binarias a nivel de usuario.

Volvamos al manejo de la complejidad: a las interfaces bien definidas. Éstas permiten a los diseñadores trabajar más o menos independientemente. Por ejemplo, el ISA es una de esas interfaces. El software puede correr en diferentes plataformas que implementen el mismo ISA, y un hardware que implemente dicho ISA podrá correr ese software. Las interfaces bien definidas permiten el desarrollo en distintas compañías y en distinto momento de subsistemas que interaccionarán entre sí.

A pesar de sus muchas ventajas, las interfaces bien definidas pueden también ser limitantes. Los subsistemas diseñados para las especificaciones de una interfaz no funcionarán con los diseñados para otras. Los programas de aplicación están ligados a un conjunto de instrucciones y sistema operativo específicos. Un sistema operativo está ligado a unas interfaces de E/S y sistema de memoria específicas.

Además de la interfaz hardware/software, las cuestiones sobre la gestión de los recursos hardware también pueden limitar la flexibilidad del sistema. Los sistemas operativos desarrollados para una arquitectura específica (uniprocesador o multiprocesador de memoria compartida) están diseñados para gestionar los recursos directamente. Esto implica que los recursos de un sistema son gestionados por un único SO. Esto limita la flexibilidad del sistema, no sólo en términos de las aplicaciones software disponibles, sino en términos de seguridad y aislamiento de fallos, especialmente cuando el sistema es compartido por múltiples usuarios o grupos de usuarios.

La virtualización proporciona una manera de relajar las restricciones anteriores y aumentar la flexibilidad. Cuando un sistema o subsistema es virtualizado, su interfaz y todos los recursos visibles a través de ella son mapeados en la interfaz y recursos de un sistema real que verdaderamente los implementa. En consecuencia, el sistema real se transforma, pareciendo ser un sistema virtual distinto o incluso un conjunto de múltiples sistemas virtuales.

Formalmente, la virtualización implica la construcción de un isomorfismo que mapea un sistema *huésped* a un *anfitrión* real. Este isomorfismo, mostrado en la Figura 2.1, mapea el

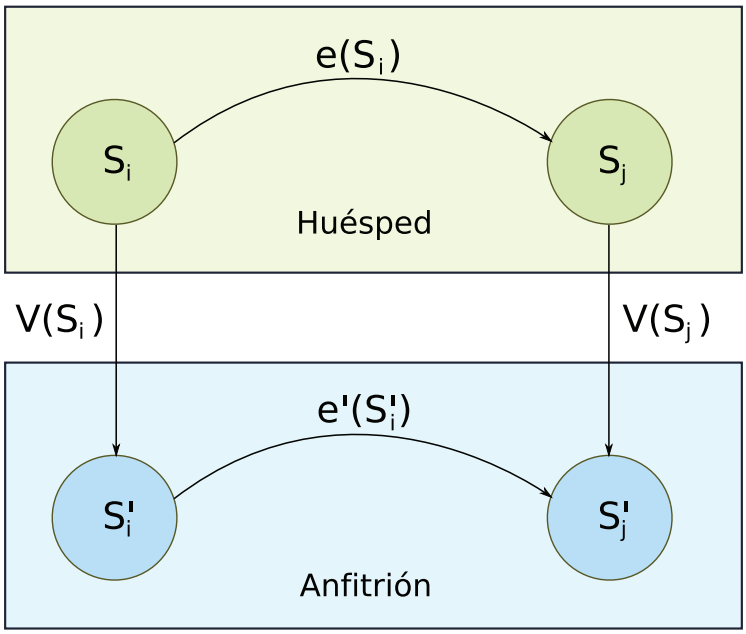

Figura 2.1: Virtualización. Formalmente, la virtualización es la construcción de un isomorfismo entre un sistema huésped y un anfitrión; $e' \circ V(S_i) = V \circ e(S_i)$.

estado del huésped al estado del anfitrión (función V en la Figura 2.1), y para una secuencia de operaciones, e, que modifica el estado en el huésped (la función e modifica el estado S_i al estado S_j) hay una secuencia de operaciones e' en el anfitrión que realiza una modificación equivalente en el estado del anfitrión (cambia S'_i a S'_j). Aunque tal isomorfismo puede usarse para caracterizar tanto abstracción como virtualización, existen diferencias entre las dos: virtualización se distingue de abstracción en que virtualización no necesariamente oculta detalles; el nivel de detalle en un sistema virtual suele ser el mismo que en el sistema real subyacente. La virtualización se puede aplicar a susbistemas, como discos, o a la máquina completa.

Existe una amplia variedad de máquinas virtuales para proporcionar una igualmente amplia variedad de beneficios. Se pueden implementar múltiples máquinas virtuales replicadas en una única plataforma hardware para proporcionar a cada usuario o grupo de usuarios sus propios entornos de sistemas operativos. Los diferentes entornos de sistema (posiblemente con sistemas operativos distintos) también proporcionan aislamiento y seguridad mejorada. Un gran servidor multiprocesador puede dividirse en servidores virtuales más pequeños, manteniendo la capacidad de balancear el uso de los recursos hardware a través del sistema. La máquinas virtuales también pueden usar técnicas de emulación para soportar compatibilidad multi-plataforma, haciendo que software escrito para una plataforma se ejecute en otra. Además de la emulación, las máquinas virtuales pueden proporcionar optimizaciones dinámicas, al vuelo, de los binarios de los programas. Además, con emulación, las máquinas virtuales pueden habilitar nuevos conjuntos de instrucciones, usando por ejemplo una arquitectura VLIW mientras se soportan los programas existentes con el conjunto de instrucciones estándar.

Los ejemplos descritos en los párrafos anteriores se construyen para coincidir con arquitecturas de máquinas reales existentes. Pero también hay máquinas virtuales sin máquina real correspondiente. Es común inventar máquinas virtuales a la medida de nuevos lengua-

jes de alto nivel. Los programas escritos en estos lenguajes de alto nivel se compilan en binarios dirigidos a la máquina virtual. Entonces cualquier máquina real para la que exista una implementación de la máquina virtual puede ejecutar el código compilado. Un ejemplo claro es Java y su máquina virtual.

2.5. Conceptos básicos de virtualización

Para entender lo que es una Máquina Virtual (MV), primero hay que aclarar qué entendemos por máquina. Para un proceso ejecutando un programa de usuario, la máquina consiste en un espacio de direcciones lógicas de memoria asignado al proceso, junto a los registros e instrucciones a nivel de usuario que permiten la ejecución del código del proceso. La parte de E/S de la máquina sólo es visible a través del SO. Por tanto la máquina es una combinación de SO y el hardware a nivel de usuario subyacente. El ABI proporciona la interfaz entre el proceso y la máquina (Figura 2.2a).

Para el sistema operativo, la máquina está implementada únicamente por el hardware subyacente, y el ISA proporciona la interfaz entre el sistema y la máquina (Figura 2.2b). El sistema operativo gestiona los recursos del hardware y soporta al sistema completo.

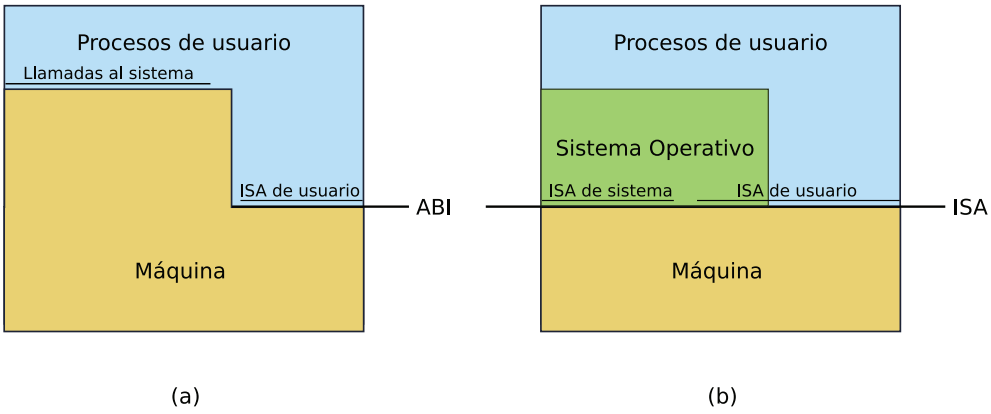

Figura 2.2: Interfaces de máquinas. (a) ABI: Interfaz Binaria de Aplicación; (b) ISA: interfaz de la arquitectura del conjunto de instrucciones.

En términos prácticos, la MV ejecuta software (ya sea un proceso o un sistema completo) de la misma manera que la máquina para la que el software se desarrolló. La máquina virtual se implementa como una combinación de máquina real y software de virtualización. La MV puede tener diferentes recursos que la máquina real, en cantidad o en tipo.

Al igual que la máquina real, donde vemos que existe una perspectiva de proceso y una de sistema, existen máquinas virtuales a nivel de proceso y a nivel de sistema. Como indica su nombre, una MV de proceso es capaz de soportar un proceso individual. En este caso el software de virtualización se sitúa sobre el ABI, encima de la combinación SO/hardware. El software de virtualización emula tanto las instrucciones de nivel de usuario como las llamadas al sistema operativo.

Con respecto a la terminología, nos referimos a la plataforma subyacente como el *anfitrión*, y al software que se ejecuta en el entorno de la MV como el *huésped*. La plataforma real que se corresponde con una máquina virtual (la máquina emulada por la MV) es la máquina nativa. El nombre dado al software de virtualización depende del tipo de máquina virtual siendo implementada. En el caso de MV de proceso este software es conocido como "runtime software".

Por el contrario, una MV de sistema proporciona un entorno de sistema completo. El entorno puede sorportar un sistema operativo junto a sus potenciales muchos procesos de usuario. Da soporte al SO huésped, dándole acceso a los recursos del hardware subyacente.

El software de virtualización en este caso se encuentra entre el hardware y el software convencional. En las MVs de sistema, el software de virtualización es conocido como el VMM (*Virtual Machine Monitor*).

El software de virtualización puede aplicarse de varias formas para conectar y adaptar los principales susbistemas del computador. La emulación añade una flexibilidad considerable permitiendo portabilidad multiplataforma, por ejemplo emulando un ISA en otro. La virtualización puede mejorar la emulación usando optimizaciones. También puede proporcionar replicación de recursos, por ejemplo, dando a una única plataforma de hardware la apariencia de múltiples plataformas capaces de ejecutar un SO completo cada una.

2.6. Maquinas virtuales de proceso

Las MVs a nivel de proceso proporcionan a las aplicaciones de usuario un entorno ABI. En sus varias implementaciones pueden proporcionar replicación, emulación y optimización.

2.6.1. Multiprogramación

La primera y más común MV es tan obvia que ni pensamos que lo sea. Muchos sistemas operativos pueden soportar múltiples procesos de usuario mediante multiprogramación, dando a cada proceso la impresión de que tiene una máquina completa para sí mismo.

2.6.2. Emuladores y traductores binarios dinámicos

Un problema mayor para una MV de nivel de proceso es soportar programas binarios compilados para un conjunto de instrucciones diferente al ejecutado por el hardware del anfitrión. El sistema operativo sobre el que se ejecuta el programa binario puede ser el mismo que aquel sobre el que se ejecuta normalmente, aunque compilado para distinta arquitectura. La forma más directa de realizar la emulación es mediante *interpretación*. Un intérprete ejecutando el ISA destino lee, decodifica y ejecuta cada instrucción fuente binaria. Esto puede ser un proceso muy lento, requiriendo decenas de instrucciones destino para cada instrucción fuente ejecutada.

Para mejor rendimiento, suele usarse la traducción binaria. Bloques de instrucciones fuente se convierten en instrucciones destino que realizan las funciones equivalentes. Puede haber una alta sobrecarga asociada al proceso de traducción, pero una vez que un bloque se tradujo, las instrucciones traducidas pueden ejecutarse muchas veces mucho más rápidamente que interpretándolas. Así funcionan los llamados traductores binarios dinámicos.

2.6.3. Optimizadores de binarios de mismo ISA

La mayoría de traductores no sólo traducen de código fuente a destino, sino que también hacen optimizaciones de código. Esta idea nos conduce también a MVs en que las que el ISA del anfitrión y el huésped es el mismo, y la optimización del programa binario es el principal objetivo de la MV. Un ejemplo es Dynamo [4], obra de HP, que, en esencia, interpreta el ISA PA-8000 de HP para ejecutarlo sobre un procesador también PA-8000.

2.6.4. Maquinas virtuales de lenguajes de alto nivel. Independencia de la plataforma

Para las MVs de proceso descritas anteriormente, la portabilidad multiplataforma es claramente un objetivo importante (poder ejecutar programas que usan un ISA sobre una máquina con otro ISA diferente). Sin embargo, sólo permiten compatibilidad caso por caso y requieren una MV para cada par de plataformas. El problema es peor aún si consideramos que las plataformas anfitrionas pueden ejecutar distintos sistemas operativos que aquél para el que los binarios fueron originalmente compilados.

La compatibilidad multi-plataforma se consigue más fácilmente dando un paso atrás y diseñándola dentro de una infraestructura de software general. En este caso, el entorno de la MV no se corresponde con ninguna plataforma real. En su lugar, está diseñada para facilitar la portabilidad y coincidir con las características de un lenguaje de alto nivel usado para el desarrollo de las aplicaciones. Estas MVs de lenguaje de alto nivel son similares a las MVs de proceso descritas antes. Sin embargo, minimizan las características específicas del hardware y del SO ya que éstas comprometen la independencia de plataforma.

En lugar de distribuir un programa en forma de binario, se distribuye un código máquina abstracto, similar a código intermedio, que es el código máquina para la máquina virtual. Tenemos por tanto un ISA virtual. Para cada plataforma, tendremos una MV capaz de ejecutar el ISA virtual. Ejemplos de esto son la arquitectura de la maquina virtual de Java o la infraestructura de lenguaje común (CLI) de Microsoft, base de .NET.

2.7. Máquinas virtuales de sistema

Estas máquinas virtuales proporcionan un entorno de sistema completo en el que muchos procesos, probablemente pertenecientes a múltiples usuarios, pueden coexistir. Usando MVs de sistema, una plataforma hardware anfitriona puede dar soporte simultáneo a múltiples sistemas operativos huésped.

Las dos implementaciones más importantes de este tipo de máquinas virtuales son:

- Virtual Machine Monitor (VMM) situado directamente sobre el hardware, y las MVs sobre él. El VMM se ejecuta en el modo de mayor privilegio, mientras que los sistemas huéspedes se ejecutan con menos privilegios. Entonces, de una manera completamente transparente, el VMM intercepta e implementa todas las acciones del SO del huésped que interactúan con los recursos del hardware. Esta es la implementación más eficiente.
- Software de virtualización colocado sobre un sistema operativo anfitrión, resultando en lo que se llama *hosted* VM. Una gran ventaja es que el SO proporciona los *drivers* de los dispositivos y otros servicios de bajo nivel que ya no deben ser proporcionados por el VMM. La desventaja es que puede perderse algo de eficiencia por las capas de software añadidas al usar un SO. Esta es la aproximación usada por productos como VMware.

También existe una aproximación, usada por plataformas de virtualización como Xen, llamada paravirtualización. Usando la paravirtualización, el sistema operativo es consciente de que se está ejecutando sobre una máquina virtualizada, no sobre la máquina real, y gracias a ello podemos evitar la pérdida de rendimiento que surge en la virtualización tradicional debido a que debemos ocultar al SO que se ejecuta en una máquina virtual. La desventaja de la paravirtualización es que para poder ser usada no nos sirve el SO original, sino que el SO huésped debe estar específicamente modificado para ejecutarse de manera consciente sobre la máquina virtual. Esto hace que los sistemas operativos de código propietario deban ser modificados expresamente por su fabricante para poder usar paravirtualización con ellos.

Vemos a continuación distintos tipos de máquinas virtuales de sistema.

2.7.1. Emulación. Máquinas virtuales de sistema completo

En las MV de sistema descritas anteriormente, todo el software de sistema (del anfitrión y del huésped) usaban el mismo ISA como hardware subyacente. Sin embargo puede que tengamos software que usa un ISA distinto (por ejemplo, el de un servidor SPARC y el de una máquina x86). Surgen las llamadas máquinas virtuales de sistema completo. El software de virtualización debe emular el entorno hardware completo. Lo más normal es situar el VMM sobre un SO anfitrión que se ejecute sobre el hardware. El VMM debe emular todas las instrucciones, convirtiendo las operaciones del ISA del sistema huésped a llamadas al SO del sistema anfitrión equivalentes.

2.7.2. Codesigned virtual machines

Estas MVs se diseñan para poder probar ISAs innovadores o implementaciones de hardware que buscan rendimiento o consumo energético mejorados. El ISA del huésped puede ser totalmente nuevo, o puede estar basado en uno ya existente al que añade o elimina instrucciones. No hay, por tanto, aplicaciones nativas para ese ISA. En este caso, será como si

el software de virtualización fuese parte de la implementación hardware, y se encargará de traducir el nuevo ISA en un ISA ya conocido para el que sí existen aplicaciones. Un ejemplo de codesigned VM es el Transmeta Crusoe. En este procesador, el hardware subyacente es un VLIW, con su ISA propio, mientras que el ISA huésped, visible al exterior, es el Intel IA-32 (más conocido como x86). Gracias al uso de la arquitectura VLIW conseguimos reducir el consumo, y la MV codesigned se encarga de la traducción de un ISA a otro.

2.8. Razones para virtualizar

Consolidación de servidores

Los grandes servidores son máquinas caras, ya que deben satisfacer unos requisitos de fiabilidad, seguridad y refrigeración muy elevados. Cuando se usan para servidores de bases de datos, suele haber otras capas computacionales, por ejemplo para ejecutar la lógica del negocio y lanzar las peticiones a la base de datos que se encuentra en el servidor. Estas capas suelen encontrarse en otros servidores más pequeños, que añaden complejidad a la gestión y mantenimiento de la infraestructura. Sería deseable poder llevar todos estos pequeños servidores dentro del gran servidor en que se encuentra la base de datos, ahorrándonos costes tanto en hardware como en gestión y mantenimiento.

Otra situación similar se da si tenemos una máquina por cada usuario o grupo de usuarios. Si pudiésemos agruparlos todos en un único servidor se reducirían los costes asociados a la gestión y mantenimiento. Sin embargo, los usuarios pueden ser reticentes a hacerlo por motivos de seguridad, o simplemente porque usan SOs distintos. Virtualizando conseguimos satisfacer todas sus necesidades.

Mejorar la utilización del sistema

La mayoría de sistemas se diseñan para satisfacer los picos de carga, aun cuando la carga media típica sea sólo una fracción de los picos. La posibilidad de particionar el sistema y ejecutar simultáneamente un número variable de cargas de trabajo permite que el sistema se configure según las necesidades cambiantes de estas cargas. Esto es más barato que tener que adquirir y gestionar un servidor para cada carga. Así conseguimos que las necesidades de cómputo globales se acerquen más a la suma de las cargas medias que a la suma de los picos de carga.

Aislamiento de fallos

Una de las principales razones de la popularidad del particionado es su capacidad para aislar los fallos. Los riesgos son los ataques por la red, fallos del software e incluso fallos del hardware. Un SO puede aislar los fallos de un proceso para que no afecten a los demás. Pero conforme aumenta la complejidad de los SOs es más fácil que se produzcan fallos en el mismo. El particionado aisla los fallos a la partición en que se produjeron.

Se puede discutir que el particionado podría llevarlo a cabo el SO, de manera que los recursos se repartieran entre los procesos y existiera realmente protección entre ellos. Sin embargo esto no proporciona MVs para ejecutar varios SOs distintos, algo que puede ser necesario.

2.9. Relación entre simuladores y virtualización

Si nos fijamos, un simulador puede considerarse como una máquina virtual: da la impresión al software simulado de que se está ejecutando sobre una máquina real. Si bien un simulador de manera estricta sería un sistema de virtualización, ni el propósito ni los posibles usos de la simulación y la virtualización, tal y como suelen entenderse, son los mismos.

El objetivo de los simuladores es el ayudar en la investigación: proporcionan mecanismos para obtener estadísticas sobre el rendimiento y permiten modificar los parámetros de la máquina simulada. La virtualización, por el contrario, pretende aprovechar al máximo los recursos físicos disponibles, proporcionándonos la mayor flexibilidad en su utilización (permitiéndonos, por ejemplo, instalar varios SOs en una misma máquina real).

Además, vimos que la velocidad de un simulador suele ser muy lenta, fácilmente varios órdenes de magnitud más lenta que la máquina real. En contraposición, uno de los objetivos comunes al usar virtualización es obtener un rendimiento todo lo cercano posible al de la máquina real.

3

Simics: elementos y programación del simulador

Como vimos en la sección 2.3.2, Simics [14] es un simulador de sistemas completos que proporciona un nivel de detalle tal que puede ejecutar un sistema operativo sin modificar. Vamos a ver los elementos que componen al simulador y que nos permiten extenderlo para, a partir de ellos, poder construir nuestro propio módulo que modele la temporización del sistema de memoria de la máquina simulada por Simics.

3.1. Módulos en Simics

La funcionalidad de Simics puede extenderse con módulos escritos por el usuario. Hay dos tipos de módulos en Simics: *dispositivos*, que modelan el comportamiento funcional de dispositivos reales, y *extensiones*, que extienden Simics de otras maneras. Un tercer tipo de módulo sería el módulo de procesador, pero el usuario no puede crear módulos de este tipo. No hay una distinción en la práctica entre dispositivos y extensiones; es posible crear módulos que tanto modelan dispositivos como extienden Simics de otras formas. También es posible modelar varios dispositivos en el mismo módulo implementando varias clases en él (clases de Simics, no confundir con las del lenguaje de programación). Realmente, un módulo de Simics puede hacer prácticamente cualquier cosa que queramos.

Los módulos se suelen escribir en C o C++, pero también puede usarse Python para la mayoría de cosas. Python es un buen lenguaje para realizar prototipos rápidos de los módulos, pero para obtener un buen rendimiento es preferible usar C/C++.

Sólo hay dos funciones que un módulo deba contener. Son **init_local()** y **fini_local()**. Son llamadas cuando un módulo se carga en Simics y cuando se elimina, respectivamente. Estas funciones permiten al módulo hacer lo que necesite en esos momentos. Simics no permite todavía eliminar módulos, así que fini_local() suele estar vacía.

Un módulo de Simics puede hacer cualquier cosa. Puede incluso lanzar un nuevo hilo que ejecute un programa totalmente diferente dentro de Simics. Para ser de algún uso práctico, sin embargo, un módulo normalmente debe interaccionar con Simics, con otros módulos, o

con el usuario. El API de Simics proporciona mecanismos que pueden usarse con este fin. Los más importantes son las clases y objetos, interfaces, haps y eventos.

3.1.1. Clases y objetos

La configuración del sistema simulado por Simics viene determinada en términos de objetos y atributos. Cada dispositivo está representado por un objeto, cuyos atributos se corresponden con el estado del dispositivo.

En la práctica, la mayoría de los módulos de Simics funcionan de la misma forma; crean objetos que proporcionan nueva funcionalidad a través de sus atributos y comandos. Los objetos de Simics, como es lógico, son instancias de las clases de Simics. Una clase describe el tipo de objeto y sus atributos e interfaces. Simics maneja la traducción entre un objeto de Simics y un objeto en Python automáticamente, de manera que los atributos del objeto de Simics puedan accederse como en un objeto en Python normal. En C pueden usarse las funciones SIM_get_attribute() y SIM_set_attribute() para manipular los atributos de los objetos.

3.1.2. Interfaces

Cuando se necesita una interacción entre objetos más compleja que la que es posible con atributos, Simics nos permite el uso de interfaces. Una interfaz especifica un conjunto de funciones que deben ser proporcionadas por cualquier clase que implemente dicha interfaz. Una clase puede implementar una interfaz para hacerse accesible a otras clases, y también puede acceder a otras clases a través de las interfaces que éstas implementen. Además, las interfaces proporcionan un mayor nivel de abstracción, de forma que la implementación de la interfaz es independiente de la interfaz en sí. Una de las interfaces más importantes es la interfaz *io-memory*, implementada por aquellos dispositivos que pueden ser mapeados en un espacio de memoria. Como todos esos dispositivos implementan la misma interfaz, el espacio de memoria puede tratarlos de la misma manera. Otras interfaces pueden usarse para cosas como lanzar interrupciones del procesador o implementar funcionalidad de un dispositivo PCI.

Una interfaz se implementa básicamente como una estructura de punteros a funciones. Cada clase que implementa una interfaz proporciona su propio conjunto de funciones requeridas por la interfaz, y entonces registra esta implementación de la interfaz en Simics usando la función *SIM_register_interface()* .

3.1.3. Eventos

A veces un módulo necesita ejecutar determinado código cuando haya transcurrido cierto tiempo. Esto puede conseguirse añadiendo un evento, que hará que una función *callback* se llame cuando haya transcurrido una cantidad de tiempo determinada. En Simics hay dos colas de eventos, la cola de tiempo, que mide el tiempo simulado en ciclos, y la cola de pasos, que mide los pasos simulados. La diferencia entre ciclos y pasos puede verse en la sección 3.3.

Los dispositivos generalmente usarán la cola de tiempo, salvo que estén interesados en la cola que mide el número exacto de pasos de la CPU. Un dispositivo que desee lanzar una interrupción de reloj cada 10ms es un ejemplo de módulo que debe usar la cola de tiempo.

3.1.4. Haps

Un módulo también puede querer reaccionar ante ciertos eventos de Simics. Estos pueden ser eventos relacionados con la máquina simulada, como excepciones del procesador o escritura en registros; u otro tipos de eventos, como que Simics detenga la simulación y vuelva a la línea de comandos. En Simics dichos eventos se conocen como *haps*. Para reaccionar ante un hap, el módulo debe registrar una función *callback* para dicho hap. Esta función se llamará cada vez que el hap tenga lugar.

3.1.5. Comandos

Un módulo que interaccione con el usuario debe crear comandos, que serán invocados por el usuario (o por un script) desde la línea de comandos de Simics. Los comandos pueden estar ligados tanto a una clase como al espacio de nombres global. Los comandos ligados a una clase sirven para manipular las instancias individuales de dicha clase. Por ejemplo, la mayoría de dispositivos tienen un comando *status* que imprime el estado del dispositivo. Los comandos globales, por el contrario, se usan para cosas que no pertenecen a un objeto en concreto.

Las clases implementadas por un módulo, los atributos del mismo, las interfaces que implementa, etcétera, son registradas en Simics mediante las funciones del API, como SIM_register_class() para el caso de las clases. De este modo Simics tiene conocimiento de los módulos que existen, puede interactuar con ellos y puede permitir al usuario que indique los módulos, su configuración y las conexiones entre ellos que desea usar en la simulación.

3.2. Interfaz de Jerarquía de Memoria

Tanto la CPU como los dispositivos pueden iniciar transacciones de memoria en Simics. Para la CPU, la dirección virtual solicitada es traducida en primera instancia por la MMU (*Memory Management Unit*), y se realiza un acceso a memoria usando la dirección obtenida. Las transacciones de los dispositivos normalmente se saltan la fase de traducción, pero por lo demás son tratadas de la misma forma.

Para saber donde se encuentra en realidad una dirección física, Simics usa un concepto llamado espacio de memoria, al que ya hemos hecho mención. Un espacio de memoria mapea direcciones físicas a objetos que pueden aceptar transacciones de memoria, como la RAM, una memoria flash o un dispositivo. Un acceso realizado al espacio de memoria es automáticamente propagado al destino correcto (dispositivo o memoria).

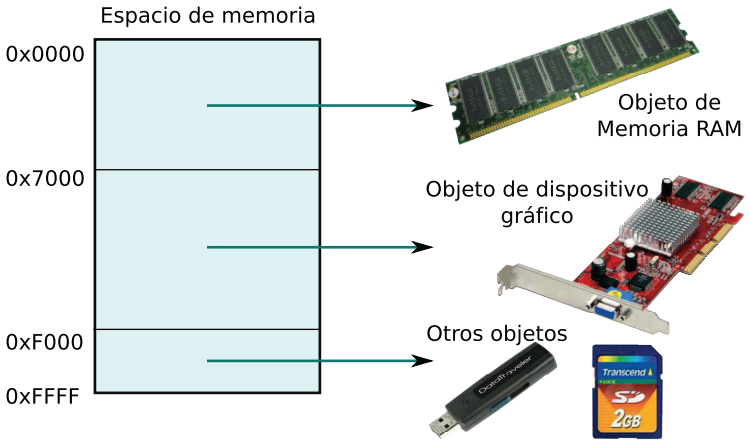

Figura 3.1: Ejemplo de espacio de memoria en Simics. *Los objetos de cualquier clase que implemente una interfaz de jerarquía de memoria pueden pertenecer a un espacio de memoria.*

Los espacios de memoria pueden también contener otros espacios de memoria, que a su vez mapearan la dirección física a un objeto específico, creando así una organización jerárquica de espacios de memoria. Esto se suele usar para separar distintos tipos de mapeo o implementar diferencias entre arquitecturas.

Los espacios de memoria proporcionan una *interfaz de jerarquía de memoria* para poder observar y modificar las transacciones que pasan por ellos. Esta interfaz está realmente compuesta por dos interfaces distintas, que actúan en dos fases diferentes de la ejecución de la transacción de memoria:

- La interfaz de modelo de temporización (*timing-model interface*) proporciona acceso a la transacción antes de que sea ejecutada (cuando acaba de llegar al espacio de memoria). Para conectar un objeto a la interfaz *timing-model* debemos asignar dicho objeto al atributo *timing_model* del espacio de memoria correspondiente. Al construir nuestro módulo, en la propia inicialización -función init_local()- realizaremos esta asignación para que las peticiones de memoria de la máquina simulada atraviesen nuestro módulo, pudiendo de esa manera trabajar con ellas. La interfaz *timing-model* puede usarse para cambiar la temporización y ejecución de una transacción de memoria, así como para modificar el valor de un almacenamiento que va hacia memoria. GEMS, el simulador de sistemas de memoria, la usa para controlar el tiempo que tarda una transacción de memoria en resolverse.
- La interfaz de *snoop-memory* proporciona acceso a una transacción después de que haya sido ejecutada. La ventaja de esta interfaz es que los valores de las cargas de memoria son accesibles, ya que la operación ya se ha realizado. Esta interfaz también puede usarse para modificar dichos valores.

Ambas interfaces (*timing-model* y *snoop-memory*) pueden usarse simultaneamente, incluso por el mismo objeto. En nuestro caso tan sólo nos interesa el interfaz *timing-model*. Veremos como implementar esta interfaz cuando desarrollemos nuestro primer módulo para Simics en la sección 3.4.

3.2.1. Deteniendo las transacciones de memoria

Cuando un objeto conectado a la interfaz *timing-model* recibe una transacción de memoria, está autorizado a modificar la temporización de la transacción devolviendo un tiempo de detención (en adelante *stall time*), en número de ciclos de procesador. Si se recibe un *stall time* distinto de cero, Simics bloqueará la transacción durante ese número de ciclos del procesador antes de ejecutarla.

Este comportamiento es clave para modelar cachés y jerarquías de memoria en Simics, especialmente en el caso de simulaciones multiprocesador.

Para mejorar la velocidad de la simulación, Simics no realiza todos los accesos a través del espacio de memoria. Para ello usa la llamada Cache de Traducción del Simulador (STC), que debemos desactivar para poder modelar la temporización de *todos* los accesos, ya que en caso contrario habrá accesos que se resuelvan directamente en la STC sin que podamos intervenir en su ejecución.

El proceso que Simics sigue para resolver un acceso de memoria (teniendo en cuenta que para nuestros propósitos la STC está desactivada) es el siguiente:

1. La CPU ejecuta una instrucción de acceso a memoria.
2. La transacción de memoria se crea.
3. Si un modelo de temporización está conectado con el espacio de memoria, recibe la transacción.
    a) Si el modelo de temporización devuelve un *stall time* distinto a cero, el procesador se detiene y la transacción se volverá a ejecutar (es decir, se enviará de nuevo al modelo de temporización) cuando el tiempo de detención haya terminado.
    b) Si el modelo de temporización devuelve un *stall time* de cero, el espacio de memoria es libre para ejecutar la transacción.
4. El espacio de memoria determina el objeto destino (normalmente la RAM).
5. El objeto recibe la transacción y la ejecuta.
6. Si un objeto snoop-memory está conectado, recibe la transacción.
7. La transacción vuelve a la CPU con los datos correctos.

En nuestro caso nos interesa intervenir en el paso tres. Es en dicho paso en el que GEMS establece el tiempo que tarda en ejecutarse cada acceso a memoria. Para ello hay que llevar, obviamente, un control de todas las peticiones que hay activas en dicho instante. Además debemos simular toda la jerarquía de memoria, y en su caso la red de interconexión entre procesadores, los protocolos de coherencia, etcétera, necesarios para simular de la forma más detallada posible una transacción de memoria. De todo ello se encargará GEMS.

3.3. Pasos y ciclos en Simics

Simics es un simulador dirigido por eventos con una resolución máxima de un ciclo de reloj. En un sistema monoprocesador, un ciclo se define fácilmente como uno dividido por la frecuencia del procesador establecida por el usuario al configurar la simulación, complicándose algo más esta definición en caso de un multiprocesador.

Conforme Simics hace que el tiempo simulado avance, ciclo por ciclo, los eventos se disparan y se ejecutan. Los eventos incluyen interrupciones de dispositivos, actualización del estado interno, así como ejecución de "pasos". Un "paso" es la unidad en que Simics divide la ejecución de un flujo de instrucciones que fluye a través de un procesador; un paso se define como la ejecución de una instrucción, de una instrucción resultado de una excepción, o de una interrupción. Los "pasos" son de lejos el más común de los eventos.

Los "pasos" y los ciclos son conceptos fundamentales de Simics. Simics muestra (y permite registrar) dos tipos de eventos a los usuarios, los ligados a un paso específico y los ligados a un ciclo específico. Para cada ciclo ejecutado, los eventos se disparan en el siguiente orden:

1. Todos los eventos agregados para ese ciclo específico, excepto los "pasos".
2. Para cada paso planificado para ejecutarse en ese ciclo:
   a) Todos los eventos planificados para ese paso específico.
   b) La ejecución del paso en sí.

Los eventos que pertenecen a la misma categoría se ejecutan en orden FIFO: el primero que se registró se ejecuta primero.

3.4. Un módulo que implementa la interfaz timing-model

Usaremos los elementos que nos proporciona Simics, descritos en el Capítulo 3.1, para desarrollar el módulo de Simics inicial que nos servirá de base para abordar el desacoplamiento entre Simics y GEMS.

Para controlar la latencia de los accesos a memoria nos interesa implementar la interfaz timing-model de Simics en nuestro módulo, tal y como hace GEMS para conectarse con Simics. Esta interfaz es extremadamente sencilla. Tan sólo consta de una función que recibe como parámetro una estructura de datos que representa un acceso a memoria y contiene toda la información relacionada con el acceso. Esta función, que implementaremos en nuestro módulo, debe devolver el valor correspondiente al número de ciclos de reloj de Simics que la operación de memoria tarda en resolverse. Es por tanto nuestro módulo quien se encarga de calcular la temporización de los accesos de memoria y de pasar dicha información a Simics como respuesta al envío por parte de éste de las peticiones de memoria a nuestro módulo.

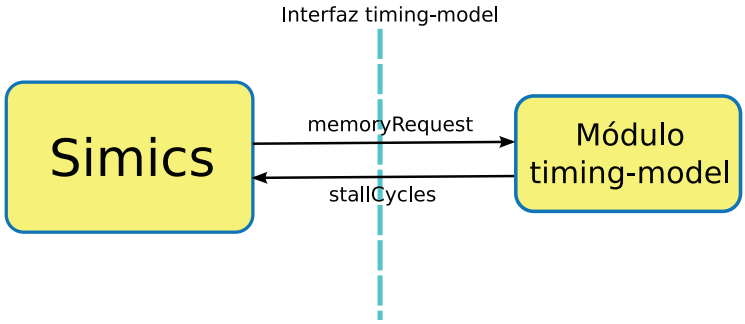

Figura 3.2: Interfaz timing-model. *Módulo que implementa la interfaz timing-model de Simics y recibirá las peticiones de memoria*

Podemos observar la estructura de esta comunicación en la Figura 3.2. Nuestro módulo recibirá el nombre de **mi-device**.

Simics puede simular un entorno multiprocesador, y redirige a nuestro módulo mi-device, una por una y en el orden temporal en que se producen, las peticiones de memoria de todos los procesadores. Una petición de memoria recibida de Simics no indica en qué ciclo de reloj se produjo. Cuando mi-device responde con un número de ciclos a una petición de memoria, esa cantidad va dirigida exclusivamente al procesador que realizó la petición, de entre todos los simulados por Simics.

Los procesadores proporcionados por Simics son procesadores con ejecución en orden, no segmentados, con accesos a memoria bloqueantes, y que por defecto completan una instrucción en cada ciclo de reloj. Por ello, cuando una instrucción emite una petición a memoria, ya sea para buscar la propia instrucción, para acceder a un dato, o para escribirlo, el cauce del procesador se bloquea hasta que el acceso a memoria se haya resuelto.

Cuando un procesador de Simics que ha realizado una petición recibe como respuesta un valor distinto de cero, dicho procesador se bloquea hasta que haya transcurrido el número de ciclos que se indicó que debe tardar la petición de memoria. Una vez haya transcurrido ese número de ciclos, el procesador volverá a realizar de nuevo la misma petición de memoria. Este proceso se repetirá hasta que el procesador reciba un valor de cero ciclos, momento en que podrá continuar con su ejecución. El bloqueo que se produce cuando el número de ciclos devuelto es distinto a cero es del procesador que emitió la petición; en una simulación multiprocesador, el bloqueo de uno de los procesadores no afecta a los demás, que pueden seguir lanzando peticiones. Esto no significa que no pueda darse el caso en que todos los procesadores se lleguen a bloquear, al recibir cada uno de ellos un valor de ciclos de espera distinto a cero, en respuesta a sus peticiones de memoria.

4

GEMS

El simulador GEMS se encarga, básicamente, de modelar la temporización de un sistema de memoria. GEMS es un módulo de Simics que implementa la interfaz *timing-model*. Dadas las peticiones de memoria de una máquina simulada por Simics, GEMS indica el tiempo que tardan en resolverse en el sistema de memoria. GEMS tiene un módulo principal, llamado Ruby, que se encarga de modelar el sistema de memoria en sí. También se compone de un conjunto de módulos auxiliares, llamados *Drivers*, que son los encargados de enviar las peticiones de memoria y otra información al módulo Ruby, y de recibir las indicaciones de Ruby cuando las peticiones de memoria se hayan resuelto.

4.1. Driver de Ruby

GEMS nos proporciona una interfaz para poder desarrollar nuestros propios Drivers que se comuniquen con Ruby. Para ello debemos usar la interfaz determinada por la clase abstracta Driver. Esta es la misma interfaz que usan los Drivers que ya existen en GEMS.

El Driver tiene dos funciones básicas. La primera es la de recibir las peticiones desde su origen, y pasárselas a Ruby para que se ejecuten en el sistema de memoria. Esto se debe implementar con la función *makeRequest* del Driver. La segunda función del Driver es recibir de Ruby el aviso de que una petición de memoria, que el Driver le pasó previamente a Ruby, se ha resuelto. Esto se implementa mediante la función *hitCallback*, que es llamada por Ruby cuando se resuelve una petición y recibe los datos necesarios para saber de qué petición de las realizadas se trata.

4.2. Drivers existentes en GEMS

Ruby proporciona varios Drivers (elementos que dirigen su ejecución) que pueden servir como origen de las solicitudes de operaciones de memoria para Ruby, los principales son:

Tester: Se usa para para probar el funcionamiento de Ruby. Normalmente, cuando desarrollamos un nuevo protocolo de coherencia de caché para GEMS, usamos el *Tester*, para probar que no se producen interbloqueos o para obtener trazas de la ejecución del protocolo y poder comprobar su correcto funcionamiento.

Simics: Este Driver usa el simulador funcional de Simics para aproximar un procesador simple, en orden, sin detenciones en el cauce. Este es el tipo de procesador proporcionado por Simics, como se indicó en el Capítulo 3.4. Simics pasa a Ruby todas las solicitudes de carga (load), almacenamiento (store) y búsqueda de instrucciones (instruction fetch). Ruby lleva a cabo el acceso al primer nivel de caché para determinar si la operación acierta o falla en la caché primaria (normalmente L1). Si hay acierto, Simics continua ejecutando instrucciones, cambiando entre los procesadores si se está simulando un multiprocesador. Si por el contrario hay un fallo, Ruby detiene la petición y simula el fallo de caché. Como ya hemos comentado, los procesadores simulados por Simics son procesadores en orden que por defecto ejecutan una instrucción por ciclo. Cuando se detiene una petición de un procesador, el procesador en sí se detiene hasta que se haya resuelto el fallo de caché. Por tanto, cada procesador puede tener un solo fallo de caché pendiente de resolver, y la contención y otras interacciones temporales entre los procesadores determinarán cuando se resuelve la petición. Al controlar la faceta temporal, decidiendo cuando los procesadores de Simics avanzan, Ruby determina la simulación funcional dependiente del tiempo en Simics (por ejemplo, para resolver cual es el siguiente procesador en adquirir un bloque de memoria).

Opal: Este Driver modela un SPARC v9 dinámicamente planificado y usa a Simics para verificar su corrección funcional. Envía sus peticiones de memoria a Ruby.

El diseño modular de GEMS proporciona una gran flexibilidad. Por ejemplo, el simulador del sistema de memoria (Ruby) es independiente del simulador del procesador fuera de orden (Opal). Un investigador puede desarrollar una mejora del sistema de memoria y obtener resultados preliminares usando el modelo simple de procesador en orden de Simics, que es mucho más rápido que Opal. Tras ello puede decidir usar el simulador fuera de orden.

GEMS también proporciona flexibilidad a la hora de especificar los diferentes protocolos de coherencia de caché que se pueden simular en el simulador de temporización. Los detalles dependientes del protocolo están separados de los componentes y mecanismos del sistema independientes del protocolo. Para facilitar la especificación de nuevos protocolos se proporciona un lenguaje de especificación de protocolos llamado SLICC.

Lo más importante es que los dos módulos principales de GEMS son Ruby (modela el sistema de memoria) y Opal (modela un procesador SPARC v9 fuera de orden que puede usarse como Driver de Ruby).

4.3. Ruby

Ruby es un simulador de un sistema de memoria de un multiprocesador que modela: cachés, controladores de caché, red de interconexión del sistema, controladores de memoria y bancos de la memoria principal.

Ruby está implementado en C++ y usa un modelo de eventos, que es gestionado mediante una cola, para simular la temporización. Los componentes se comunican usando buffers de mensajes de latencia y ancho de banda variables. Se planifica al componente en el extremo receptor del buffer para que despierte cuando el siguiente mensaje esté disponible para ser leído del buffer. Aunque muchos buffers se usan en un estricto orden FIFO, no están restringidos a este comportamiento. La simulación se lleva a cabo invocando el método para despertar al siguiente evento planificado de la cola de eventos. Conceptualmente, la simulación sería idéntica si todos los componentes se despertaran cada ciclo, pero la cola de eventos puede verse como una optimización para evitar procesamiento innecesario cada ciclo.

4.3.1. Componentes independientes del protocolo

Los componentes independientes del protocolo de coherencia de Ruby incluyen la red de interconexión, los arrays de las cachés, los arrays de la memoria, los buffers de mensajes, y lógica variada.

La red de interconexión se usa para comunicar los controladores de caché y memoria. Se usa una red simple monolítica para simular todas las comunicaciones, incluso entre controladores que estarían en el mismo chip en un sistema CMP (MultiProcesador en un Chip) simulado. Toda la comunicación intra-chip e inter-chip se maneja como parte de esta interconexión, aunque cada enlace individual puede tener distintos parámetros de latencia y ancho de banda. Este diseño proporciona suficiente flexibilidad para simular la temporización de casi cualquier sistema.

Ruby modela una red de interconexión conmutada punto a punto que puede configurarse de forma similar a la red de interconexión de los sistemas multiprocesadores actuales, incluyendo tanto aquellos sistemas basados en directorio como los basados en protocolos de snooping. Entre las topologías posibles tenemos redes punto a punto totalmente conectadas, toros 2D con encaminamiento dinámico, redes totalmente ordenadas (por ejemplo *crossbars*). Además pueden especificarse nuevas topologías mediante ficheros de texto donde indiquemos los enlaces, sus latencias, ancho de banda, etcétera.

4.3.2. Funcionamiento de Ruby

El objetivo de Ruby es, como ya se indicó, el de determinar la *latencia* de los accesos a memoria.

Un elemento muy importante en el interior de Ruby es el *Sequencer*. Cada procesador simulado tiene asignado un Sequencer dentro de Ruby. A éste será a quien se le comuni-

quen las peticiones del procesador. Es importante darse cuenta de que tenemos una "doble" simulación de los procesadores: por una parte, en Simics, estamos simulando el comportamiento funcional del procesador, mientras que en Ruby estamos simulando el tiempo de resolución de sus accesos a memoria. Hay una equivalencia por tanto entre los procesadores simulados en Simics con los simulados en Ruby. Las peticiones de memoria de un procesador concreto de Simics se corresponden con un procesador concreto de Ruby. Por tanto, las peticiones que Ruby reciba de un procesador de Simics debe enviárselas al Sequencer que corresponde a dicho procesador. El Sequencer representa al procesador dentro del sistema de memoria, y será el encargado de lanzar la petición desde el lugar dentro del sistema que le corresponde al procesador (es decir, desde la ubicación exacta del procesador dentro de la red de interconexión). La petición recorrerá el camino que deba a través de la jerarquía de memoria (caché L1, caché L2, directorio, memoria DRAM, etcétera), atravesando todos los enlaces de interconexión que sea necesario (y en el instante en que sea posible), hasta resolverse.

El proceso seguido para establecer la latencia de las peticiones de memoria es un tanto peculiar. Como puede verse en el Capítulo 3.2.1, Simics realiza una solicitud pasando la petición de memoria al módulo que implementa la interfaz "timing-model". En este caso se trata del módulo Ruby. Como vimos en aquel capítulo, Simics espera que se devuelva a su petición el número de ciclos que el procesador que la lanzó debe estar detenido hasta que la petición se haya resuelto. Una vez hayan pasado esos ciclos, el procesador que se detuvo realiza de nuevo la misma petición de memoria. El proceso se repite hasta que el procesador reciba un valor igual a cero, lo que significa que la petición se resolvió y por tanto puede continuar con su ejecución. No hay que olvidar que el módulo que implementa la interfaz "timing-model" tan sólo calcula el tiempo que tardan en resolverse las peticiones: la funcionalidad, es decir, la carga o almacenamiento de los datos en memoria, se sigue llevando a cabo en Simics.

Dentro de Ruby tenemos un módulo encargado de controlar las interacciones entre Ruby y Simics. Éste módulo es conocido por Multifacet (los creadores de GEMS) como Simics, y en la sección 4.2 nos referíamos a él también con ese nombre, es decir, con el mismo nombre que el simulador Simics. Este módulo formaría con el propio simulador Simics el *Driver* que realiza las peticiones al sistema de memoria modelado por Ruby. Esto es útil porque hay peticiones de Simics que no deben pasar al sistema de memoria, por diversas razones que veremos más adelante, y éste módulo se encarga de gestionar y filtrar los accesos a memoria que deben acceder al sistema de memoria, y tratar directamente aquellos que no deben hacerlo. Llamaremos a este módulo SimicsRuby para evitar confusiones con el simulador Simics.

En la Figura 4.1 podemos observar como quedan encuadrados en el esquema los módulos GEMS, Ruby y SimicsRuby.

La cuestión interesante es que, cuando recibimos una solicitud, no podemos saber cuando se resolverá, ya que nos encontramos en un sistema de memoria de un multiprocesador, y el valor de ciclos que deseamos conocer puede depender de otras peticiones que aún no se han realizado. Para comprender esto, basta con imaginarse un procesador que realiza una petición de lectura de un bloque de memoria que no tiene en caché. Supongamos que otro procesador, que se encuentra más cercano al directorio asignado a dicho bloque, realiza una petición de escritura varios ciclos después. Al estar más cerca, la solicitud de éste último

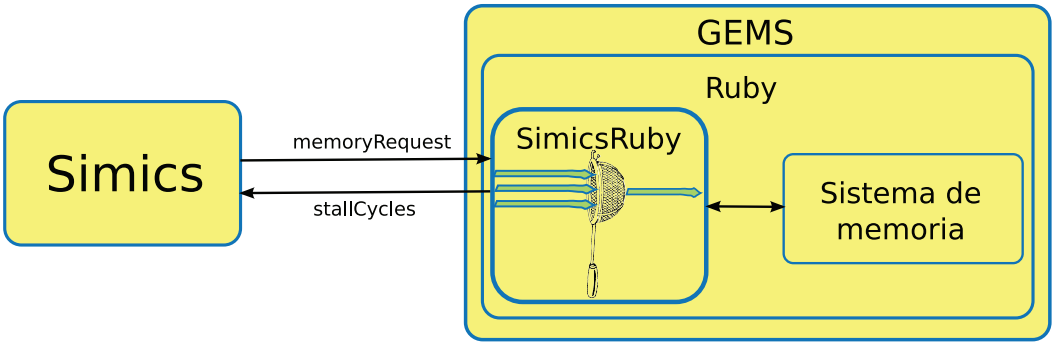

Figura 4.1: Esquema de Ruby. *El módulo principal de GEMS es el simulador de sistemas de memoria Ruby. Ruby a su vez contiene el módulo SimicsRuby que se usa como Driver para el Sistema de Memoria simulado por Ruby.*

procesador llega antes y obtiene el bloque en un estado exclusivo. Cuando llegue la solicitud del primer procesador, deberán realizarse acciones adicionales de coherencia para obtener el bloque de la caché del otro procesador, que lo modificó, incrementándose por tanto la latencia del acceso a memoria del primer procesador. Acabamos de ver una situación en que la latencia de un acceso depende de sucesos posteriores al momento que se produjo la petición. Por tanto, ¿cómo podemos devolver una latencia a Simics cuando uno de los procesadores realiza una petición de memoria? Existirían varias soluciones posibles. La adoptada en Ruby consiste en devolver a cada petición que pasa al sistema de memoria un valor de ciclos extremadamente alto (concretamente dos mil millones de ciclos) de manera que estemos seguros de que el procesador permanecerá bloqueado mientras no se haya resuelto la petición. Conforme avance la ejecución de Ruby (que como vimos era dirigida por eventos) se irán resolviendo las peticiones que hayan pendientes.

Pero entonces, ¿cómo hacemos para que el procesador despierte una vez que su petición se haya resuelto? El API de Simics nos proporciona una función, SIM_stall_cycle(proc_ptr, num_cycles), a la que podemos llamar pasándole como parámetros el procesador al que queremos "detener" y el número de ciclos que queremos detenerlo. El truco está en que si el número de ciclos de detención que indicamos es cero, en lugar de detener al procesador, lo que conseguimos hacer es despertar a un procesador que se encontraba detenido. De esa manera, cuando se resuelva un acceso a memoria de un procesador al que hemos bloqueado durante dos mil millones de ciclos, podremos despertarlo.

Sin embargo aparece una nueva duda. Si Simics llama al módulo que implementa la interfaz *timing-model* únicamente para pasarle las peticiones, tras lo cual el control del programa pasa de nuevo a Simics, ¿cómo es posible que llamemos a SIM_stall_cycle en el ciclo exacto en que debemos despertar al procesador detenido?. Es más, ¿cómo es posible que se sincronicen Simics y Ruby para que los ciclos que simula uno coincidan con los que simula el otro?. En principio la única información que Ruby tiene es la que recibe cuando le pasan las peticiones de memoria, y en ella ni siquiera se encuentra información sobre el ciclo en que se realizó la petición. Llegados a este punto es el momento de abordar un tema que se ha dejado de lado hasta ahora por simplicidad en la explicación de los módulos. Nos referimos a la **sincronización** entre Simics y el módulo que implementa la interfaz *timing-model*, en este caso Ruby. Más adelante veremos como nosotros hacemos lo propio para nuestro módulo mi-device. Necesitamos que la ejecución en los dos simuladores,

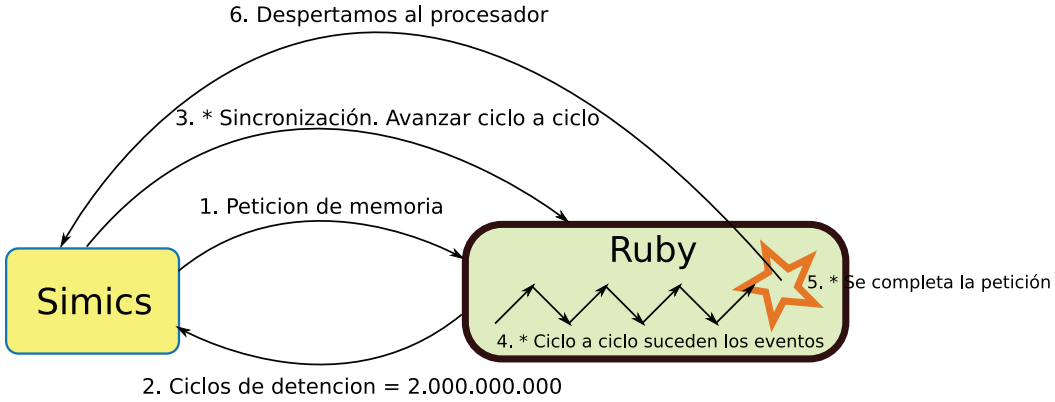

Figura 4.2: Sincronización de Simics y Ruby. *Vemos como primero se produce la petición de memoria (1), a la que Ruby responde deteniendo el procesador durante un número enorme de ciclos (2). Cada callback (3) hace que Ruby avance un ciclo y se lancen los eventos correspondientes a ese ciclo. La simulación del sistema de memoria va avanzando (4) hasta que se resuelve el acceso realizado por el procesador (5). Entonces Ruby despierta al procesador detenido (6), que puede continuar con su ejecución y lanzando nuevas peticiones de memoria.*

Simics y Ruby, avance sincronizada, de manera que los ciclos en uno y otro simulador vayan sucediéndose acompasadamente.

La solución la encontramos de nuevo en el API de Simics. Nos proporciona una función, SIM_time_post_cycle(), que añade un evento a la cola de tiempo. Con esta función registramos un *callback* a una función, e indicamos que se llame dentro de un número fijo de ciclos. Cuando se ejecute dicha función, Ruby sabrá que ha transcurrido dicho número de ciclos en Simics, consiguiendo así una forma muy sencilla de sincronización. Además, en ese momento Ruby ejecutará los eventos de su cola de eventos que se corresponden con ese ciclo. Conforme se lancen estos eventos se irán resolviendo las peticiones a memoria de los distintos procesadores, y se llamará a la función que sirve para despertarlos, SIM_stall_cycle. De esa manera, un acceso a memoria se resuelve en el número de ciclos que va desde que se produjo la petición del procesador, hasta que se despertó al procesador con este mecanismo que acabamos de explicar.

Al final de la ejecución de esta función *callback*, para que se ejecute de nuevo se volverá a registrar con SIM_time_post_cycle(). El número de ciclos que transcurren entre cada *callback* viene determinado por la variable de configuración de Ruby SIMICS_RUBY_MULTIPLIER, usada como parámetro de la función que registra el *callback*. Como Simics modela unos procesadores en orden muy sencillos, que ejecutan una instrucción por ciclo y sólo pueden tener una petición a memoria en ejecución, usando este "multiplicador" conseguimos simular de manera aproximada un procesador de mayor complejidad que puede enviar varias peticiones en un mismo ciclo. Por tanto, un ciclo de Ruby se corresponde con uno o varios ciclos de Simics.

El esquema de la sincronización puede observarse en la Figura 4.2.

Cuando usa a Simics como Driver, como en este caso, Ruby (concretamente el módulo SimicsRuby) debe gestionar las peticiones pendientes de cada procesador. Debido al modelo de procesador usado en Simics, un procesador sólo puede tener una petición a memoria

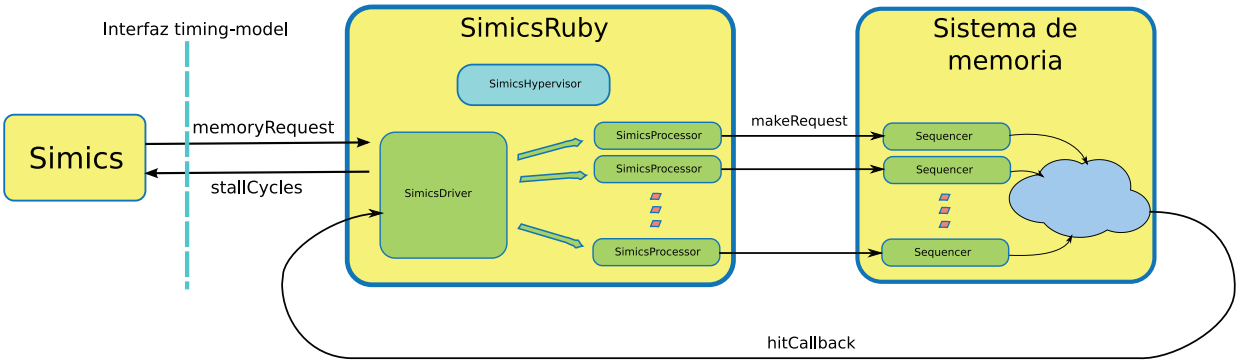

Figura 4.3: Estructura del módulo SimicsRuby. *Vemos los tres elementos del módulo SimicsRuby: el Hypervisor, el Driver y los Processors. Cada processor se corresponde con un Sequencer del sistema de memoria, al que envía las peticiones que pasan el filtro.*

pendiente. Sin embargo, cuando el procesador sea despertado con el procedimiento indicado anteriormente, dicho procesador volverá a realizar la misma petición, como ya vimos, y por tanto es necesario que el módulo de Ruby que se encarga de interactuar con Simics, SimicsRuby, sea capaz de reconocer la petición como la misma que acaba de resolverse para ese procesador, y devolver ahora un valor de cero ciclos directamente, sin volver a ejecutar la petición, para que el procesador continúe con su ejecución.

Como cada procesador sólo puede tener una petición pendiente de resolver, el número máximo de peticiones simultaneas será igual al número de procesadores de la máquina simulada por Simics.

El módulo SimicsRuby consta de los siguientes elementos:

SimicsHypervisor: Se encarga de las tareas de más alto nivel relacionadas con la comunicación entre Simics y Ruby.

SimicsDriver: Se trata de la clase que implementa la interfaz Driver, explicada en el Capítulo 4.1. Recibe las peticiones y se encarga de comprobar si la petición recibida debe ser enviada al sistema de memoria o no, y de pasarla al SimicsProcessor correspondiente. A su vez recibirá de Ruby los *callbacks* cuando se completen las peticiones de memoria, y los pasará a los SimicsProcessor correspondientes.

SimicsProcessor: Tenemos uno por cada procesador simulado. Se encargan de controlar si la petición recibida de Simics ya fue realizada con anterioridad y por tanto ya se resolvió. En caso contrario realizarán la petición al Sequencer correspondiente a este procesador. Cuando reciben un *callback*, se encargan de despertar al procesador correspondiente en Simics.

La estructura del módulo SimicsRuby puede observarse en la Figura 4.3.

5

Desacoplamiento de Simics y GEMS

5.1. Elementos de la solución

Tenemos dos nuevos elementos en la solución, que vienen a sustituir al módulo al que llamábamos SimicsRuby, que se encargaba de gestionar las peticiones de memoria que provenían de Simics y pasarlas a Ruby, así como de despertar a los procesadores de Simics cuando se resolvían los accesos a memoria. Dividiendo este módulo en dos conseguiremos desacoplar Simics y GEMS. Los dos módulos resultantes son los siguientes:

- Un módulo de Simics, llamado **mi-device**, que se encarga de llevar a cabo todas las interacciones con Simics que realizaba el módulo SimicsRuby.
- Un módulo unido a Ruby, llamado **RemoteRuby**, que se encarga de interactuar con Ruby.

Estos dos módulos deberán comunicarse entre sí de manera que el resultado de su funcionamiento sea exactamente el mismo que obteníamos usando el módulo SimicsRuby.

El esquema del sistema quedaría tal y como aparece en la Figura 5.1.

Una vez hayamos realizado esto, será posible sustituir el simulador Simics por otro con tan solo construir el equivalente a mi-device para el otro simulador. Esto nos permitirá usar GEMS para simular la temporización del sistema de memoria para otros simuladores distintos a Simics.

5.2. Modulo mi-device

Éste es el módulo que se encarga ahora de recibir las peticiones de memoria de Simics. Al conectarnos a una máquina de Simics debemos tener en cuenta que ésta puede estar simulando una máquina multiprocesador, y por tanto podemos recibir hasta una petición por cada procesador de la máquina simulada. Además, al igual que sucedía en SimicsRuby, debemos controlar qué peticiones pasarán al sistema de memoria y cuales no.

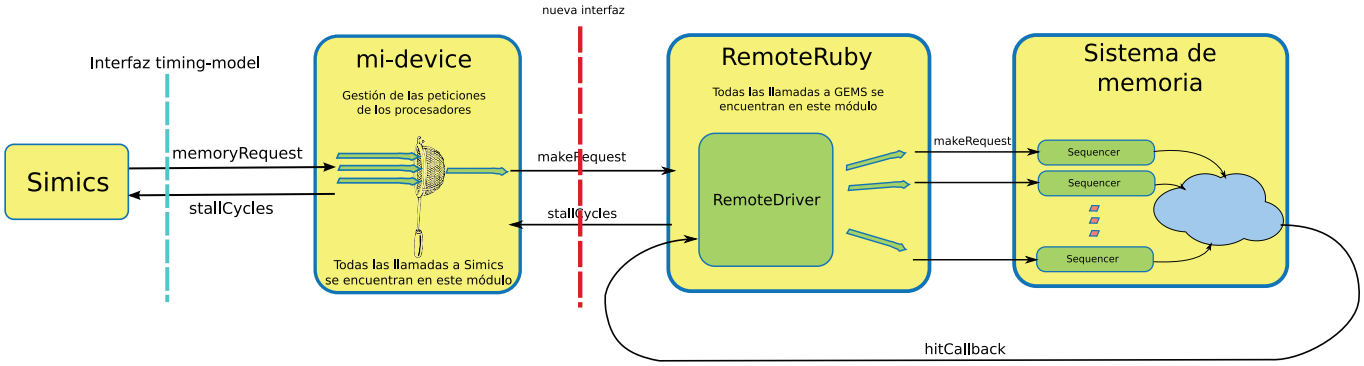

Figura 5.1: División de SimicsRuby en mi-device y RemoteRuby. *mi-device recibe las peticiones de Simics y las filtra. Se comunica con RemoteRuby, que se encarga de pasar las peticiones a los Sequencers y de recibir los hitCallbacks indicando que una petición se ha resuelto. La nueva interfaz que aparece entre mi-device y RemoteRuby separa dos procesos del sistema operativo distintos: cada lado de la interfaz se corresponde con uno de ellos.*

Una cuestión fundamental a la hora de realizar la división del módulo SimicsRuby en un módulo conectado a Simics y otro conectado a Ruby es que todas las llamadas al API de Simics deben quedar en el módulo que está conectado a Simics, es decir, en mi-device. Por tanto, nos interesa que todas aquellas operaciones que necesiten llamar al API de Simics queden en este módulo, para evitar así tener que enviar información desde RemoteRuby a mi-device y viceversa, simplificando al máximo la interfaz entre ambos módulos. De hecho, para desacoplar GEMS de Simics, de manera que podamos usar GEMS en combinación con otros simuladores, necesitamos que todo aquello que dependa de Simics quede en el módulo mi-device, que es el módulo "dependiente del simulador". En consecuencia, la interfaz entre mi-device y RemoteRuby sólo incluirá aquellas operaciones relacionadas con el funcionamiento del sistema de memoria, y en ningún caso operaciones relacionadas con el simulador concreto (Simics en este caso) que se está utilizando.

5.3. Manejo de las peticiones de memoria

Hemos comentado que no todas las peticiones de memoria se pasan al subsistema de memoria de Ruby. Es hora de ver cuales son las peticiones que no deben pasar y el motivo. Los siguientes casos son los más importantes:

- Una petición que ya se completó en memoria. Cuando un procesador que se encontraba detenido en espera de que se sirviera su petición de memoria es despertado por mi-device, inmediatamente volverá a realizar la misma petición. Así lo estipula el funcionamiento de Simics, y por ello debemos controlar este caso para que, cuando recibamos una petición repetida, directamente devolvamos un valor de cero ciclos. Así el procesador que la realizó continuará con su ejecución.
- Peticiones no detenibles. Simics define que hay ciertas peticiones de memoria para las que no podemos detener al procesador que las realiza. Esto se indica en la estructura de datos de la petición que se recibe de Simics. Ante una petición de este tipo, necesariamente hay que devolver cero ciclos, o la simulación de Simics puede fallar.

- *Prefetches*. Cuando se leen datos de memoria por adelantado, SimicsRuby por defecto devuelve un valor de cero ciclos. Nosotros imitaremos este comportamiento.
- *Flush* de Caché. Hay peticiones que únicamente sirven para limpiar la caché. Ante una petición de este tipo también devolveremos un valor de cero ciclos.

Además, se puede controlar si se pasan a memoria todas las peticiones, o si, por el contrario, no se pasan las peticiones en modo privilegiado (también llamado supervisor), y tan sólo se consideran las realizadas en modo usuario. Igualmente, podemos controlar si sólo pasamos las peticiones de datos, o si también enviamos al sistema de memoria las peticiones para traer instrucciones.

Existen varios tipos de peticiones distintas para Ruby. Son las siguientes:

- Un *instruction fetch*. Se trata de una petición para la lectura de instrucciones.
- Un *load* o carga.
- Un *store* o almacenamiento.
- Una operación atómica. Se trata de accesos a memoria que deben realizarse de manera atómica. En principio puede ser también de cualquier tipo de los anteriores, pero a Ruby sólo le interesa saber que es atómica.

Una petición que sí pasa al sistema de memoria, será enviada por mi-device a RemoteRuby, que nos responderá con el número de ciclos que el sistema de memoria simulado por Simics considera que debe detenerse. Este valor es el que mi-device devolverá a Simics.

5.4. Módulo RemoteRuby

Este módulo se conecta a Ruby y se encarga de recibir las peticiones de memoria de mi-device y devolver el número de ciclos de detención para cada petición. Además, cuando se resuelve un acceso a memoria, de la manera que se indicó en el Capítulo 4.3.2, RemoteRuby recibe el *hitCallback* de Ruby indicando la resolución del acceso y avisa a mi-device para que despierte al procesador de Simics correspondiente.

En este módulo se encuentra el nuevo Driver que "dirige" a Ruby, al que llamamos RemoteDriver. En este caso el Driver no tiene tanta funcionalidad como tenía en el módulo SimicsRuby original, ya que esta funcionalidad se ha dividido entre RemoteRuby y mi-device. Concretamente, la parte de comprobar si la petición debe ser enviada al sistema de memoria se realiza en el módulo mi-device, evitando el envío entre mi-device y RemoteRuby de las peticiones que no necesiten pasarse al sistema de memoria.

El control de qué peticiones de cada procesador deben pasar al Sequencer correspondiente en Ruby se realiza en mi-device, y por tanto en este módulo RemoteRuby no necesitaríamos llevar ningún control adicional: tan sólo deberíamos crear la petición en forma de acceso a caché, que es como reciben los Sequencers las peticiones de los procesadores, y

pasarla al Sequencer correspondiente dentro de Ruby. Sin embargo hay una situación por la que nos va a interesar controlar en RemoteRuby en qué estado se encuentra la petición realizada por cada procesador. La situación que queremos detectar se da cuando enviamos una petición al sistema de memoria, tenemos un acierto en L1 y se permite el llamado *camino rápido* en el acceso a la caché (esto último es un parámetro de configuración de Ruby). En ese caso, la petición se resolverá de inmediato, es decir, con cero ciclos de latencia. Sin embargo, para detectar el acierto en L1 la petición se envió al Sequencer, y al haberse resuelto inmediatamente se ha llamado al *hitCallback* del RemoteDriver para indicarlo. Como vimos, cuando se completa un acceso a memoria debemos enviar a mi-device la indicación para que le diga a Simics que debe despertar al procesador que realizó el acceso, y esto se realiza concretamente en *hitCallback*, al que estamos llamando ahora para indicar que se acertó en L1.

En el código original de GEMS, cuando se daba este caso de acierto en L1, se despertaba al procesador con la función del API de Simics que sirve para tal propósito. Si pensamos un poco en esta situación, vemos que estamos en disposición de devolver cero ciclos como *stall time* al procesador que realizó la petición, ya que el acierto en L1 se produce antes de que devolvamos este tiempo de detención. Por tanto, no es necesario despertar al procesador por el acierto en L1, ya que nunca llegó a detenerse. En GEMS se realizan ambas cosas: se despierta al procesador y se devuelven cero ciclos, funcionando correctamente. Sin embargo se están realizando llamadas al API de Simics que no serían necesarias, penalizando levemente el rendimiento. En nuestro caso, al tener que comunicar los módulos RemoteRuby y mi-device, se produciría el envío de la información sobre el *hitCallback* al módulo mi-device para que despierte al procesador. Esto no nos interesa ya que los aciertos en L1 presumiblemente serán un gran porcentaje de todos los accesos, lo que haría que la comunicación entre los módulos se incrementara notablemente. Por ello, a diferencia de GEMS, nuestro código controla esta situación de manera que al producirse el acierto en L1 no se envíe el mensaje a mi-device indicando que debe despertar al procesador, sino que bastará con la respuesta al makeRequest indicando que la petición se ha resuelto en cero ciclos, con lo que el procesador que la realizó continuará con su ejecución sin detenerse.

Otra cuestión importante es el manejo de los eventos. Ruby está dirigido por eventos, y necesitamos que cada cierto tiempo se disparen los eventos correspondientes a ese instante. Para ello, como vimos en el Capítulo 4.3.2, Ruby se sincronizaba con Simics usando una función del API de éste último que hacía que pasado un número de ciclos se llamase a la función en Ruby que se encargaría de disparar los eventos correspondientes a ese instante. Ahora es mi-device quien puede registrar *callbacks* en Simics. Sin embargo, es RemoteRuby quien tiene acceso a la cola de eventos de Ruby para dispararlos. Por tanto, la comunicación debe incluir un mensaje para que cuando se ejecute el *callback* de mi-device, éste le indique a RemoteRuby que debe disparar los eventos, consiguiendo así mantener la sincronización entre Simics y Ruby a través de los módulos mi-device (conectado a Simics) y RemoteRuby (conectado a Ruby).

5.5. Comunicación entre mi-device y RemoteRuby

La comunicación la realizaremos mediante tuberías FIFO. Tendremos un FIFO en cada sentido: de mi-device hacia RemoteRuby y viceversa. Es importante que la comunicación sea lo más simple posible y represente una interfaz bien definida, clara y fácil de implementar. De esta manera la sustitución de Simics por otro simulador será sencilla al ser fácil construir el módulo equivalente a mi-device para el nuevo simulador.

Para establecer esta interfaz se ha reducido al máximo posible la información que debe enviarse por los FIFOs. Para conseguir independizar a Ruby de Simics se han tenido que desplazar todas las tareas dependientes de Simics que se realizaban en Ruby hasta el módulo mi-device, de manera que por el FIFO sólo fluya información relevante para las peticiones de memoria.

Los simuladores son programas con tiempos de ejecución especialmente altos, y es recomendable que intentemos optimizar el código lo más posible. Por ello se ha definido la comunicación de manera que tanto el envío como la recepción de un mensaje se lleve a cabo con una única escritura o lectura en el FIFO, respectivamente. El coste de esto es que todos los mensajes tendrán el tamaño del mayor de ellos. Esto es una sobrecarga importante. Sin embargo, al realizarse la comunicación a través de la memoria, y al evitar realizar una lectura adicional para leer los mensajes (evitando así llamadas al sistema operativo y cambios de contexto adicionales), la sobrecarga queda sobradamente compensada.

Primero cubriremos la comunicación básica y necesaria para el funcionamiento de la infraestructura de virtualización, y más adelante trataremos las características adicionales, como la recolección de estadísticas del simulador.

Los mensajes necesarios para la comunicación son los siguientes:

De cliente (mi-device) a servidor (RemoteRuby):

configuration: Envía la configuración del cliente. El único valor que se envía actualmente es el número de procesadores, ya que realmente no es necesaria otra información para el funcionamiento de Ruby. Se envía al inicio de la comunicación.

makeRequest: Envía una petición de memoria para que sea resuelta en el sistema de memoria. Los datos que contiene esta petición son los exigidos por Ruby para poder resolver el acceso. Estos son:

- número del procesador de la máquina simulada por Simics que realizó la petición
- dirección física dentro de la máquina simulada por Simics a la que quiere accederse
- dirección virtual a la que se accede
- contador del programa, apunta a la dirección virtual de la instrucción que está realizando el acceso
- tipo de la petición (*load*, *store*, *instruction fetch* o acceso atómico)
- tamaño de la petición

- modo de acceso (modo supervisor o modo usuario)

triggerEvents: Indica a RemoteRuby que se ha avanzado un ciclo y deben dispararse los eventos de la cola que correspondan. No contiene más información.

executionEnd: Indica a RemoteRuby que terminó la ejecución que se quería realizar de la máquina simulada y puede terminar la simulación del sistema de memoria cuando lo estime oportuno. Mientras no se reciba respuesta de parte de RemoteRuby se continuará con la simulación en Simics, a pesar de que la simulación que queríamos realizar ya terminó.

Mensajes del servidor (RemoteRuby) al cliente (mi-device):

stallCycles: Llega como respuesta a un makeRequest del cliente, e indica el tiempo que debe detenerse el procesador que realizó la petición de memoria. Como único valor lleva el número de ciclos.

triggerEventsACK: Es la respuesta al triggerEvents del cliente. Indica que Ruby ya lanzó todos los eventos y por tanto el cliente puede seguir con su ejecución. Es mediante triggerEvents y triggerEventsACK como se lleva la sincronización entre Simics y Ruby. No lleva ningún dato.

hitCallback: Es la indicación de que se resolvió una petición de un procesador. Como único dato lleva el número de procesador que realizó la petición.

executionEndACK: Indica que Ruby terminó la simulación del sistema de memoria, sirve de respuesta al executionEnd enviado por mi-device. Cuando mi-device reciba este mensaje indicará a Simics que finalice la ejecución.

La comunicación debe llevarse siguiendo las reglas que se explican a continuación:

- Cada procesador sólo puede tener una petición de memoria pendiente en cada instante. Nos aprovechamos de que los procesadores simulados por Simics tienen esta limitación.
- Cuando mi-device envía un makeRequest, RemoteRuby sólo responderá con un stallCycles.
- Cuando mi-device envía un makeRequest, no puede enviar otro mensaje hasta que haya recibido el stallCycles correspondiente.
- Cuando mi-device envía un triggerEvents, RemoteRuby puede responder con cualquier mensaje menos un stallCycles. RemoteRuby enviará un número indeterminado de mensajes a mi-device -indicando accesos a memoria resueltos o cualquier otra cosa-, y finalmente enviará el triggerEventsACK.
- Cuando mi-device envía un triggerEvents, no puede enviar otro mensaje hasta recibir el triggerEventsACK correspondiente.

- Cuando mi-device envíe un executionEnd no esperará respuesta de RemoteRuby, y continuará enviando triggerEvents (y makeRequests si así lo desea) siguiendo las reglas anteriores, hasta que llegue el executionEndACK desde RemoteRuby. En ese momento la ejecución de los dos simuladores (el del sistema completo y el del sistema de memoria) habrá terminado y mi-device ya no debe enviar más mensajes a RemoteRuby.

Basándonos en los mensajes explicados en este apartado y en las reglas que deben cumplirse podemos crear fácilmente un módulo para otro simulador distinto a Simics que haga las veces de mi-device y se conecte con RemoteRuby. De ese modo podremos usar Ruby como simulador de sistema de memoria unido a otros simuladores, cumpliendo así con uno de los objetivos de este proyecto: desacoplar Ruby y Simics. Los mensajes de la comunicación están definidos en un fichero de cabecera en C que puede usarse en la construcción de estos módulos.

5.5.1. Comunicación adicional

El uso de simuladores en investigación, como ya se comentó, sirve para obtener resultados sobre las características de la ejecución de las distintas arquitecturas que estamos simulando y sus diversos elementos, para así poder llegar a conclusiones sobre hacia donde dirigir las investigaciones, probar el impacto de nuevas ideas de diseño y decidir cual es la mejor opción de entre las planteadas.

Es necesario que el simulador ofrezca unas estadísticas lo suficientemente detalladas, y que tengamos un control adecuado sobre la extracción de las mismas para poder decidir cuales y cuando deseamos obtenerlas, para que así puedan servir al propósito por el que en ese momento estemos usando el simulador.

GEMS genera una estadísticas detalladas y proporciona los mecanismos necesarios para poder extraerlas de la forma que nos resulte más conveniente en cada instante. Necesitamos pues, no perder esta cualidad al desacoplar Simics y GEMS.

Ruby, al ser un módulo de Simics, proporciona estadísticas cuando se le ordena desde Simics. A su vez Simics le ordena esto cuando el usuario lo decide mediante los diversos métodos disponibles para hacerlo, como por ejemplo la línea de comandos de Simics. Debemos por tanto introducir en la comunicación entre mi-device y RemoteRuby los mensajes necesarios para obtener las estadísticas del sistema de memoria simulado por Ruby de la misma forma en que se obtienen cuando se usa Ruby como módulo de Simics.

Los nuevos mensajes, que se envían desde el cliente al servidor, son los siguientes:

clearStats: Indica a RemoteRuby que debe limpiar las estadísticas.

dumpStats: Indica a RemoteRuby que debe imprimir las estadísticas.

El cliente, en nuestro caso mi-device, puede enviar estos mensajes en cualquier momento, y no debe esperar respuesta del servidor.

5.6. Comandos de mi-device

La comunicación del usuario con Simics viene dada por una interfaz de línea de comandos (CLI). El usuario puede usar scripts para realizar tareas complejas, ya que además Simics usa Python como lenguaje de script.

Cada módulo de Simics puede registrar comandos para que el usuario pueda interactuar con él a través de la línea de comandos, como vimos en la sección 3.1.5. En nuestro caso, tenemos comandos para conectar con RemoteRuby, para comenzar la simulación, para indicar que la simulación ha terminado, y para la inicialización e impresión de estadísticas. Como para lanzar las simulaciones se usan unos scripts que veremos en el Anexo A, no es necesario que el usuario conozca los comandos concretos que hay que usar, ya que los scripts se encargan de ejecutarlos.

6

Virtualización con Virtual-GEMS

6.1. Primeras consideraciones

¿Cómo usar virtualización dentro de una simulación? La primera respuesta es usar software de virtualización dentro del sistema simulado. Ya que disponemos de simuladores, como Simics, capaces de simular el sistema completo, podríamos ejecutar en el propio simulador el software de virtualización que consideremos adecuado (por ejemplo Xen, o VMWare). Sin embargo, esto nos plantea varios problemas:

- El uso de simulación suele basarse en ejecutar cargas de trabajo en el simulador, como *benchmarks*, para obtener los resultados de rendimiento. Para usar un software de virtualización deberíamos adaptar dichas cargas para ejecutarse en una máquina virtual proporcionada por dicho software de virtualización. Esto incluye crear la imagen de disco para cada máquina virtual.

- Al simular, además se suelen usar *checkpoints*, esto es, imágenes del estado de la máquina simulada, justo en el instante en que la carga de trabajo va a comenzar a ejecutarse. De ese modo podemos usar el *checkpoint* tantas veces como queramos para realizar simulaciones con dicha carga de trabajo. Si usamos software de virtualización, el *checkpoint* a usar en el simulador incluye al software de virtualización y a las máquinas virtuales con las cargas de trabajo. Necesitaremos tener, por tanto, un *checkpoint* para cada combinación de cargas de trabajo que deseemos usar en las máquinas virtuales. Además, si modificamos algún parámetro del software de virtualización (como el mapeo de la memoria de las máquinas virtuales a memoria real), deberemos crear nuevos *checkpoints* para que lo reflejen. La creación de estos *checkpoints* con el software de virtualización es una operación muy costosa, ya que a su vez debemos crear la imagen de las máquinas virtuales, que son en sí nuevos *checkpoints* dentro del *checkpoint* que usamos en el simulador. Esto puede verse en la Figura 6.1a.

- Para poder ejecutar correctamente el software de virtualización dentro de una simulación necesitamos una precisión y detalle en el simulador, sobre todo en cuestiones de temporización, que ni siquiera el simulador de sistemas completos Simics, que nosotros estamos usando, es capaz de conseguir.

Por ello nos interesa tener una infraestructura que nos permita usar los *checkpoints* de las cargas de trabajo, de los que ya disponemos, de manera sencilla. Esto implicará introducir en el simulador que usemos las técnicas de virtualización que queramos utilizar. Con este enfoque, parece claro que nos estaríamos dirigiendo hacia una virtualización mediante soporte hardware, ya que el propio simulador proporcionará la ilusión a cada MV de que se ejecuta en una máquina real. No existirá una capa de virtualización software simulada, sino que esta estará dentro del propio simulador.

En el siguiente esquema se muestra la diferencia entre usar un software de virtualización en la simulación y el modelo que deseamos implementar nosotros, en el que la virtualización se encuentra en el propio simulador.

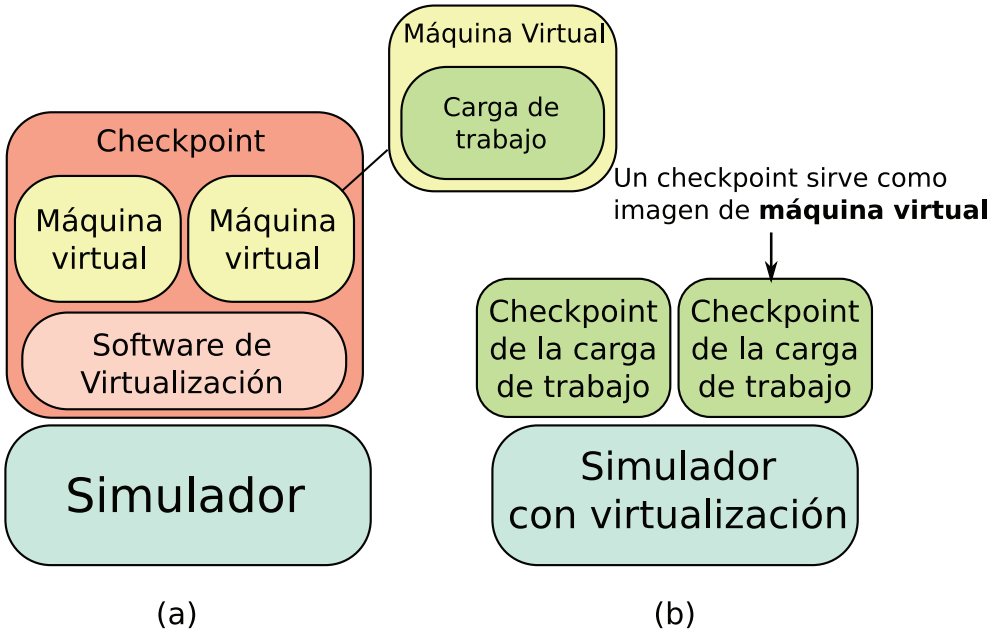

Figura 6.1: Software de virtualización simulado frente a virtualización implementada en el simulador. *(a) Usando el software de virtualización necesitamos crear un checkpoint nuevo para cada combinación de máquinas virtuales y parámetros de virtualización. (b) Usando virtualización implementada en el simulador podemos usar directamente los checkpoints de las cargas de trabajo, de los que ya disponemos, como máquinas virtuales. Podemos también variar los parámetros de virtualización en el simulador sin necesidad de crear nuevos checkpoints, y tenemos un mayor control sobre el hardware.*

Para poder usa un *checkpoint* como el de la Figura 6.1a se necesita además que cada MV se encuentre justo en el instante de ejecución que a nosotros nos interesa, con la carga de trabajo a punto para ejecutarse. Por ello, es mucho más sencillo usar el esquema de la Figura 6.1b, ya que los *checkpoints* que tenemos preparados ya se encuentran en dicho instante.

Además, la virtualización en el propio simulador nos permite un control más fino del hardware. Por ejemplo, podremos realizar el mapeo de los elementos de las MVs (procesadores, memoria, etcétera) a la máquina real de la manera que se nos antoje, de una forma mucho más flexible que la posible mediante el software de virtualización. Por ejemplo, en el Capítulo 7 veremos los distintos mapeos que realizamos de las páginas de memoria de las

MVs a la máquina real, según nos interese en cada caso, algo difícil de realizar si usáramos en nuestras simulaciones un software de virtualización ya existente.

6.2. Maquinas virtuales de sistema

Visto que implementaremos la virtualización en el propio simulador, nuestro interés se centra en las máquinas virtuales de sistema.

Un entorno de MVs de sistema es capaz de soportar varias imágenes de sistema simultáneamente, cada una ejecutando su propio sistema operativo y sus programas de aplicación asociados. Cada sistema operativo controla y gestiona un conjunto de recursos hardware virtuales, incluyendo un procesador (o procesadores), recursos de almacenamiento, y dispositivos periféricos para realizar entrada/salida para el sistema.

En un entorno de MVs de sistema, los recursos reales de la plataforma anfitriona son compartidos entre las máquinas virtuales huéspedes, con una capa de software, el monitor de máquinas virtuales (VMM), gestionando la reserva y acceso a los recursos hardware de la plataforma anfitriona. El sistema operativo de cada maquina virtual tiene la ilusión de ser dueño de los recursos y subsecuentemente los reserva para sus programas de usuario. En nuestro caso, el VMM se encontrará dentro del simulador y no existirá una capa software que realice esta labor, haciendo toda la virtualización por hardware.

Cada recurso virtual puede o no tener un recurso físico correspondiente. Si lo hay, el VMM decide como se proporcionará acceso a él a las máquinas virtuales, ya sea particionándolo o por tiempo compartido.

Algunas de las aplicaciones que justifican el uso de VM de sistema son:

Máquinas virtuales monoaplicación: Ejecutar cada aplicación en su propia MV aumenta la robustez del sistema. Un comportamiento errático de una aplicación y su SO asociado en un sistema convencional puede tirar abajo la máquina. Si esto pasa en una MV difícilmente afectará a otra aplicación que corre en otra MV. Esto es especialmente útil en nuevas aplicaciones que pueden corromper el sistema debido a bugs o virus.

Múltiples entornos seguros: Una MV de sistema proporciona una sandbox (caja de arena) que aisla un entorno de sistema de los demás, asegurando así un nivel de seguridad que no podría proporcionar un solo sistema operativo. De esta manera, los datos y actividad de un usuario no pueden ser vistos ni modificados por otro.

Entornos con varios SOs: Una sola máquina puede soportar dos SOs distintos concurrentemente. Esto soluciona desde la necesidad de usar aplicaciones que sólo están disponibles en diferentes SOs hasta la actualización a una nueva versión de SO manteniendo en otra MV la versión anterior con las aplicaciones antiguas funcionando (de igual manera nos serviría para realizar la migración de un SO a otro), pasando por situaciones en que un equipo de desarrollo realiza software que debe funcionar en varios SOs distintos. En todos estos casos nos serviría un único servidor virtualizado.

Desarrollo de software de sistema: Cuando se tienen grandes servidores, a veces es necesario desarrollar (y probar) nuevo software de sistema (por ejemplo un nuevo SO) mientras el mismo sistema está siendo usado para correr aplicaciones importantes. Usando una MV aparte para el desarrollo evitamos situaciones indeseables en que un bug en el nuevo software puede tirar el sistema abajo.

Encapsulación del sistema: Una MV de sistema proporciona una forma adecuada de encapsular el estado de la máquina completa. Esto es útil para obtener instantáneas del estado de la máquina (*checkpoints*), de manera que se pueda reanudar la ejecución desde ese punto cuando queramos, o incluso en otra máquina anfitriona distinta, permitiendo así una capacidad de portabilidad y realización de copias de seguridad muy interesante.

Nos centramos en las MV de sistema en que el ISA (conjunto de instrucciones) de anfitrión y huésped es el mismo. En nuestro caso estamos hablando de procesadores SPARC, que son los que usaremos en las simulaciones de Simics. Las peticiones de memoria que se pasan a Ruby son relativamente independientes de la arquitectura del procesador usado. Sin embargo, las peticiones emitidas por Simics sí llevan una serie de información típica de este tipo de procesador, que debemos analizar en nuestro módulo mi-device para discernir cuestiones como el tipo de petición de memoria que el procesador desea realizar (por ejemplo, un *load* o un *store*).

La virtualización de multiprocesadores es una extensión de la virtualización en monoprocesador. La diferencia es que en monoprocesador el procesador debe repartirse mediante tiempo compartido entre las MVs, mientras que con multiprocesadores puede hacerse, además, particionando los procesadores de la máquina entre las MVs.

6.3. Máquinas virtuales nativas y hosted

Un sistema de MVs en que el VMM opera en un modo de privilegio superior al modo de las máquinas virtuales huéspedes se llama sistema de MVs nativo. El VMM se instala sobre el hardware, y sobre él los SOs. El SO y aplicaciones corren en un nivel de privilegios menor, si bien el VMM puede tener que dar la impresión al SO de que se ejecuta en el mayor nivel de prioridad. Por simplicidad de implementación, el sistema de virtualización puede instalarse sobre un SO. Entonces se conoce como hosted VM system (ya que el host es el SO subyacente).

6.4. Virtualización de recursos

Los recursos gestionados por el VMM son, básicamente, tres: procesador(es), memoria y dispositivos de entrada/salida.

6.4.1. Procesador

La clave de la virtualización de un procesador está en la ejecución de las instrucciones del huésped, tanto las instrucciones a nivel de usuario como de sistema. Hay dos formas de hacerlo, mediante emulación, o mediante la ejecución nativa directa. Para este último método necesitamos que el ISA del anfitrión sea el mismo del huésped, y aun así sólo puede usarse virtualización bajo ciertas condiciones.

Usando ejecución nativa directa puede obtenerse un rendimiento cercano al de ejecutar directamente el sistema virtualizado sobre el hardware.

Un ISA bien diseñado, eficientemente virtualizable, hace que cuando una instrucción deba ser emulada ocurra un trap de manera natural, el manejador del trap salta a una rutina de interpretación adecuada, se interpreta la instrucción, y el control vuelve al programa original. Si el diseño del ISA no es bueno, habrá una sobrecarga mucho mayor para detectar qué instrucciones deben emularse.

Condiciones para la virtualizabilidad del ISA

Popek y Goldberg [22] establecieron formalmente las condiciones bajo las que un ISA puede soportar máquinas virtuales eficientemente. Parten de las siguientes suposiciones:

- El hardware consiste en un procesador y memoria direccionable de manera uniforme.
- El procesador puede operar en uno de dos modos, el modo usuario y el modo sistema.
- Un subconjunto del conjunto de instrucciones está disponible sólo en modo sistema.
- El direccionamiento de memoria se hace de forma relativa al contenido de un registro de relocalización.

El análisis podría extenderse a más de dos modos y a arquitecturas más complejas, pero los resultados permanecen inalterados. La E/S no se considera, pero el análisis puede extenderse a la E/S de manera directa.

Una instrucción privilegiada se define para nuestros propósitos como aquella que genera un trap si la máquina está en modo usuario y no lo genera si la máquina está en modo sistema. El SO que corre dentro de una MV no debe ser capaz de cambiar los recursos del sistema de modo que afecte a otras MVs o al rendimiento de las mismas. Por tanto, incluso el SO de una MV debe ejecutarse en un modo que no permita la modificación directa de recursos como el valor de temporizador de la CPU. Este valor provoca las interrupciones del temporizador cuando se agota, momento en que se ejecuta código del VMM y puede cambiar de MV en ejecución.

Para especificar las instrucciones que interactúan con el hardware, se definen dos categorías de instrucciones especiales, llamadas *instrucciones sensibles*:

Instrucciones que cambian el control: Son aquellas que intentan cambiar la configuración de los recursos del sistema.

Instrucciones de comportamiento sensible: Aquellas cuyo resultado o comportamiento depende de la configuración de los recursos.

El resto de instrucciones son consideradas "inocuas".

El teorema que nos presentan Popek y Goldberg es el siguiente: *"Para cualquier computador convencional de tercera generación, puede construirse un VMM si el conjunto de instrucciones sensibles es un subconjunto de las instrucciones privilegiadas".*

Esto significa que las instrucciones que pueden interferir con el funcionamiento correcto o eficiente del VMM (las instrucciones sensibles) siempre provocarán un trap en modo usuario. Todas las instrucciones no privilegiadas pueden ejecutarse nativamente en la plataforma anfitriona, sin necesidad de emulación. Las instrucciones sensibles fuerzan que el control vuelva al VMM, que examinará la acción deseada por la máquina virtual que lanzó la instrucción sensible y reformulará la petición en el contexto del sistema de máquinas virtuales global.

6.4.2. Memoria

Soporte a la memoria virtual en un entorno de maquinas virtuales de sistema

En un entorno de MVs de sistema, cada MV huésped tiene su propio conjunto de tablas de páginas de memoria virtual. La traducción en cada MV huésped transforma las direcciones en su espacio de direcciones virtual a lugares en la memoria real. Esta memoria real se correspondería con la memoria física en una plataforma nativa, en una plataforma de MV no es el caso. En su lugar, en un entorno de MVs de sistema, una dirección de memoria real de un huésped debe sufrir un nuevo mapeo para determinar la dirección en la memoria física del hardware anfitrión. Estamos estableciendo una clara distinción entre memoria real y memoria física, términos intercambiables en otros contextos, pero no aquí. La memoria física es la memoria del hardware. La memoria real es la ilusión de memoria física de la MV huésped; una ilusión soportada por el VMM cuando mapea la memoria real del anfitrión en memoria física. Es con este mapeo de real a física con lo que se implementa la virtualización de memoria en un sistema virtualizado. Pueden verse estos dos niveles de tablas de páginas en la Figura 6.2.

Añadir un nivel más en la jerarquía de memoria necesita mecanismos adicionales para la gestión de la memoria. El VMM debe mantener un espacio de swap debido a que la memoria total de las MVs puede ser mayor a la del sistema.

La traducción de páginas viene soportada por una combinación de tabla de páginas y una caché llamada Translation Lookaside Buffer (TLB). Dependiendo del ISA, la tabla de páginas o el TLB será arquitectural.

Si la tabla de páginas es arquitectural, su estructura está definida por el ISA, y el SO y el hardware cooperan para mantenerla y usarla. El TLB es mantenido y usado sólo por el hardware, no es visible para el sistema operativo. Cuando hay un fallo en el TLB, el hardware recorre la página de tablas arquitectural para encontrar la entrada que colocar en

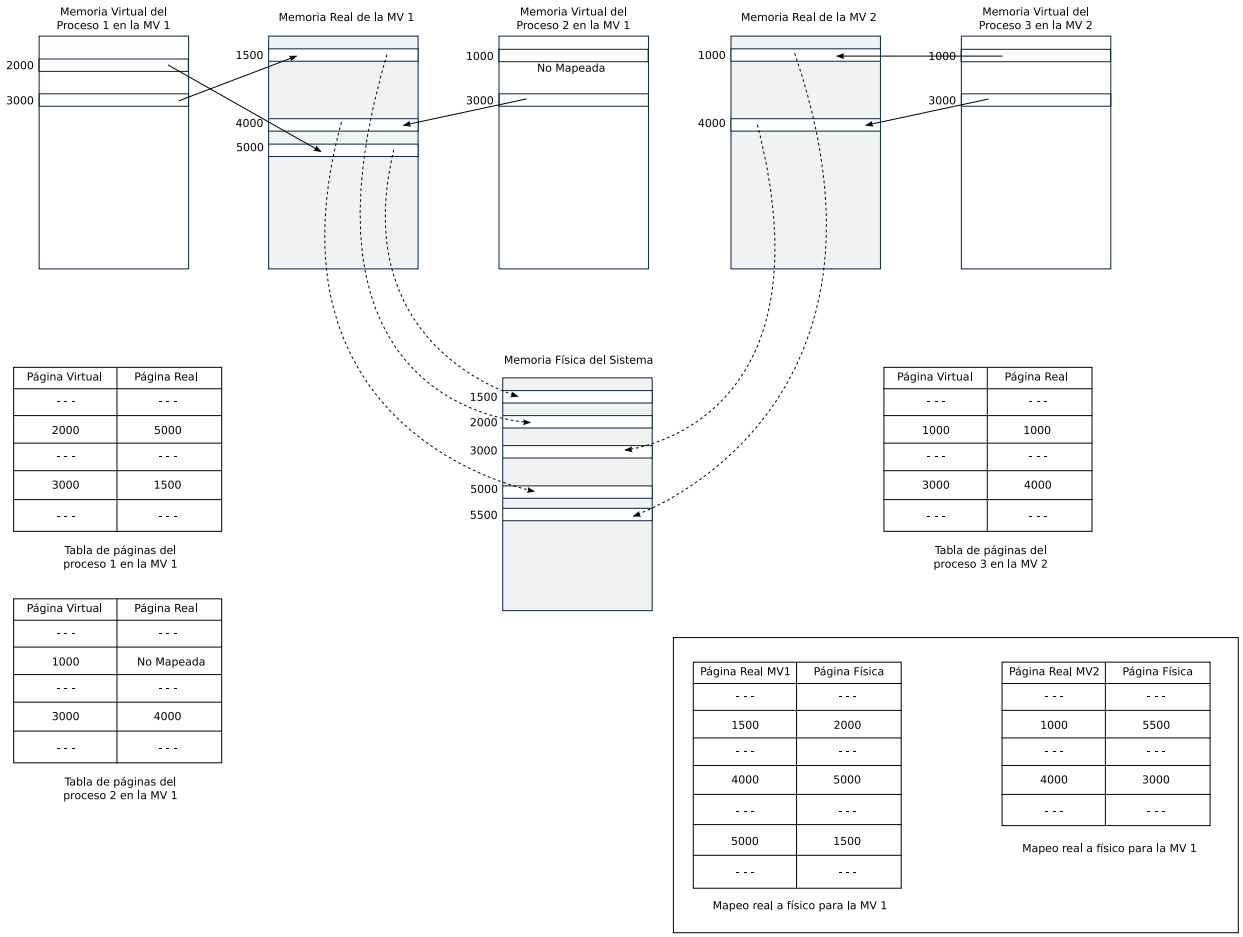

Figura 6.2: Mapeo de memoria de las MVs huéspedes. Las páginas virtuales se mapean a páginas reales por las tablas de páginas mantenidas por cada MV. Las páginas reales se mapean a páginas físicas, usando tablas de páginas mantenidas por el VMM.

el TLB; si la entrada no está mapeada a memoria física, entonces hay un fallo de página y entra en escena el sistema operativo.

Si el TLB es arquitectural, su estructura y las instrucciones especiales para manipularlo son parte del ISA. Las tablas de páginas son parte de la implementación del sistema operativo. El hardware no conoce la estructura de la tabla de páginas. De este modo, si hay un fallo de TLB, inmediatamente hay un salto al sistema operativo, que usando su tabla de páginas realiza la acción apropiada.

Virtualizando las tablas de páginas arquitecturales

El SO de cada MV mantiene sus propias tablas de páginas, una por proceso, para mapear direcciones virtuales a direcciones reales. El VMM mantiene a su vez por cada proceso de cada MV una tabla de páginas oculta que hace la traducción directa de dirección virtual a dirección física. Estas son las tablas usadas realmente por el hardware para realizar la traducción, de manera que pasamos de los dos niveles de tablas de páginas comentados anteriormente a un sólo nivel. Para conseguir esto, el registro que apunta a la tabla de páginas debe estar virtualizado, y los accesos que la MV haga a él deben provocar un trap al VMM. De esta forma, el VMM se encarga de que cada vez que el SO de una MV quiera cambiar el puntero a la tabla de páginas (para cambiar de proceso por ejemplo), el valor que tome sea el valor oculto correspondiente al proceso, y cuando quiera leerlo, le devolveremos el valor del puntero a la tabla de páginas del proceso dentro de la MV (no el de la tabla de páginas oculta). Además hay varias implicaciones a la hora de tratar los fallos de página que complican este tipo de virtualización. Por ejemplo, el VMM puede tener que hacer *swapping* debido a que la memoria de todas las MVs es mayor a la de la máquina anfitriona. Esto significa llevar una página de memoria de un proceso a disco y eliminar el mapeo a memoria de esa página de la tabla de páginas oculta del proceso. Sin embargo, ese reemplazo sólo concierne al VMM, y no a la MV. Un acceso a esa página provocará un fallo de página, pero el VMM debe resolverlo sin que la MV afectada se entere, ya que en ella no se produciría ningún fallo de página ya que realmente ella no había reemplazado dicha página.

Virtualizando un TLB arquitectural

En este caso, el SO se encarga de gestionar el TLB cuando ocurre un fallo de página. En cada entrada del TLB hay un campo que hace referencia al proceso al que pertenece la página (campo llamado ASID) de manera que varios procesos puedan tener la traducción de sus páginas en el TLB. Cuando se busca una traducción en el TLB, éste campo debe coincidir con el ASID del proceso que se está ejecutando para que la traducción sea válida para dicho proceso. Cada MV usará sus propios ASID para sus procesos, pudiendo repetirse estos entre máquinas virtuales. El VMM puede aprovechar este campo para hacer que cada proceso de cada máquina virtual tenga un ASID distinto. Necesitará una tabla para mapear los pares $(MV, ASID)$ al ASID que se usará realmente en el TLB para ese proceso. Cada vez que el SO de una MV intente escribir en el TLB la traducción de página virtual a real, el VMM intervendrá haciendo que realmente se escriba la traducción virtual a física. De este modo estaremos evitando uno de los dos niveles de tablas de páginas de la Figura 6.2.

6.4.3. Entrada/salida

La virtualización del subsistema de entrada/salida es una de las partes más difíciles de implementar un entorno de MVs de sistema. Cada dispositivo de E/S tiene sus propias características y necesita ser controlado de su propia manera. La dificultad de virtualizar la E/S viene por el hecho de que el número de dispositivos de un tipo dado suele ser muy grande y no para de crecer. Y el tipo de dispositivos también crece. La virtualización de tal número de dispositivos es un reto para cualquier VMM nuevo.

Esto es un problema igualmente para los SOs, y por eso se han desarrollado abstracciones para dar soporte a una amplia variedad de dispositivos y tipos de dispositivos. Esto también puede hacerse a nivel de VMM.

Debido a la dificultad, en nuestro caso, no virtualizaremos la E/S sino que usaremos E/S particionada, de manera que consideraremos que los dispositivos de E/S se encuentran replicados de forma que cada MV pueda tener los suyos propios, tal y como ocurre en grandes servidores que usan la técnica de particionado para lograr virtualización.

De hecho, nuestra virtualización conceptualmente se acerca más a un particionado del hardware (alcanzando la virtualización mediante mecanismos hardware) que al uso de un VMM por software. Sin embargo, el resultado alcanzado de ambas formas puede ser el mismo (máquinas virtuales de sistema), si bien en nuestro caso no estaríamos modelando las implicaciones en el rendimiento que tiene la ejecución del software de virtualización. A pesar de ello, nuestra infraestructura debería servir para obtener conclusiones que no variarían demasiado en el caso de usar un VMM por software, ya que precisamente el objetivo del VMM es afectar lo menos posible en el rendimiento y ser lo más transparente posible.

6.5. Virtualización en multiprocesadores

Hoy en día muchos sistemas, especialmente servidores y equipos de sobremesa de gama alta, contienen múltiples procesadores. Los servidores además incorporan cantidad de memoria y dispositivos de E/S. Conforme mejora la tecnología, la arquitectura multiprocesador también está abriéndose camino en portátiles y sobremesas baratos.

Normalmente no hay coincidencia entre los procesadores que idealmente necesita una aplicación y los procesadores físicos disponibles. Conforme aumenta el tamaño de los multiprocesadores, esta diferencia crece y las aplicaciones no pueden explotar todos los procesadores, ya sea por limitaciones en el paralelismo de la aplicación o por la sobrecarga por la comunicación entre procesadores. Esto ha llevado a técnicas con las que un sistema multiprocesador puede ser particionado para que varias aplicaciones puedan explotar simultáneamente los recursos. El particionado es una técnica de virtualización que podemos usar en los multiprocesadores de memoria compartida. Como resultado tendremos varios multiprocesadores virtuales conteniendo cada uno un subconjunto de los recursos de la máquina.

Antes hablábamos de la división en tiempo del uso del procesador, ahora aparece una nueva dimensión: la división en espacio. Y además ambas podrán combinarse.

6.5.1. Mecanismos para soportar particionado

Se necesita una capa entre el hardware físico y los multiprocesadores virtuales. Seguiremos llamándole VMM. El VMM puede estar implementado totalmente en hardware, en microcódigo con soporte hardware o en software con soporte hardware. Una diferencia clave es la posibilidad de tener un nuevo nivel de privilegio superior, de manera que el VMM se ejecute en él, y los SOs de las particiones puedan ejecutarse en modo sistema. De esta manera evitamos la pérdida de rendimiento que surgía cuando el SO tenía que ejecutarse en modo usuario. En nuestro caso consideramos que el VMM está implementado totalmente en hardware, al encontrarse su implementación dentro del propio simulador.

Nos centraremos en aquellas técnicas que usan soporte hardware: particionado físico y lógico.

6.5.2. Particionado físico

Se trata de la virtualización más sencilla y que menos sobrecarga introduce. Se particionan físicamente los recursos de manera que un multiprocesador virtual no pueda afectar a otro. El número de particiones máximas se reduce al número de procesadores de la máquina (o incluso a menos). El control del particionado se realiza por hardware.

Normalmente se configuran las distintas placas bases del sistema, de manera que cada multiprocesador se compone de un conjunto de estas placas, si bien suele permitirse un mapeo más fino (ya que una placa está compuesta por varios procesadores, módulos de memoria y dispositivos de E/S conectados a ella). De cualquier modo, cada procesador y cada bus de E/S pertenece a una MV. Del mismo modo, los módulos de memoria de una placa suelen pertenecer a la MV que posee dicha placa.

A pesar de sus ventajas (aislamiento de fallos, seguridad, reparto del hardware entre particiones), el particionado físico no es lo ideal para optimizar el uso del sistema. Las particiones seguirán estando infrautilizadas, por ejemplo porque la reserva de recursos para cada partición será conservadora. Además, el balanceo dinámico de los recursos no es posible debido a las restricciones físicas impuestas en pos del aislamiento de los fallos. El particionado lógico sacrifica ese aislamiento físico a cambio de una mayor flexibilidad al asignar recursos a las particiones.

6.5.3. Particionado lógico

Con este tipo de particionado el estado de dos máquinas virtuales puede estar totalmente entrelazado, no sólo en memoria sino también en los procesadores. Aparece de nuevo la compartición de procesadores de manera multiplexada por tiempo, algo que en el particionado físico no se puede realizar, ya que un procesador pertenece a una MV concreta. Este

tipo de particionado permite asignar fracciones de procesadores a las MVs, de manera que habrá procesadores asignados a una MV y otros repartidos entre dos o más MVs, y que se repartirán entre ellas por tiempo compartido (y según los pesos que tengan asignados las particiones).

De igual manera los dispositivos de E/S pueden compartirse entre MVs.

La memoria, por supuesto, puede repartirse entre los procesadores de diversas formas. Una manera sencilla de mapear los espacios de direcciones de las MV dentro de la memoria física es hacerlo de manera contigua, es decir, la memoria de cada MV tiene una dirección de inicio en memoria física y un tamaño, con lo que sólo se necesitan dos valores (puntero de inicio y tamaño) para mapear la memoria de cada MV. Con este modelo, la suma de las memorias de las MV no puede superar la memoria física del sistema.

Otra forma, usada por ejemplo en el Cellular Disco [12], es la de mapear las direcciones reales de las MV a direcciones físicas dentro del espacio de direcciones del hardware de la máquina, de la misma manera en que veíamos en la Figura 6.2.

Son varias las ventajas de este modelo. Primero, la suma de la memoria real de las MVs puede ser mayor a la memoria física (necesitaremos espacio de *swap* para el VMM). Segundo, se puede permitir la compartición de páginas entre las máquinas virtuales, bastará con que en la tabla de páginas oculta de los procesos se apunte a la misma página física. Para esta compartición, el Cellular Disco proporciona funciones para las aplicaciones mas allá de las disponibles a través de la interfaz del sistema operativo. Por ejemplo, proporciona llamadas para registrar regiones de memoria compartidas.

En la práctica, la forma de realizar la traducción de dirección virtual-real-física será una de las explicadas en el Capítulo 6.4.2.

La mayor flexibilidad que existe en el particionado lógico necesita de una gestión más compleja que el físico. Inicialmente, se implementaba el VMM en microcódigo, considerándose parte del hardware y nombrándose a menudo como firmware. Hoy en día es más común realizar el trabajo en una capa de firmware y software llamada comúnmente como *hypervisor*.

6.6. Nuestra implementación

Nuestra máquina aúna varios conceptos de las distintas técnicas de virtualización que hemos visto. En cuanto a la virtualización de procesadores, se realiza un particionado físico, con una granularidad de un procesador, de manera que podremos asignar cada procesador de la máquina física a la MV que deseemos.

En cuanto a la memoria, usamos un esquema de dos niveles de tablas de páginas, realizando el primer nivel el mapeo de dirección virtual de un proceso a dirección real de la MV, y el segundo nivel el mapeo de dirección real a dirección física de memoria en la máquina anfitriona. Conceptualmente podría considerarse como particionado lógico con una granularidad de página de memoria, ya que este segundo mapeo se realizaría a través de mecanismos hardware. Esto hace que no se consuman recursos de la máquina mediante

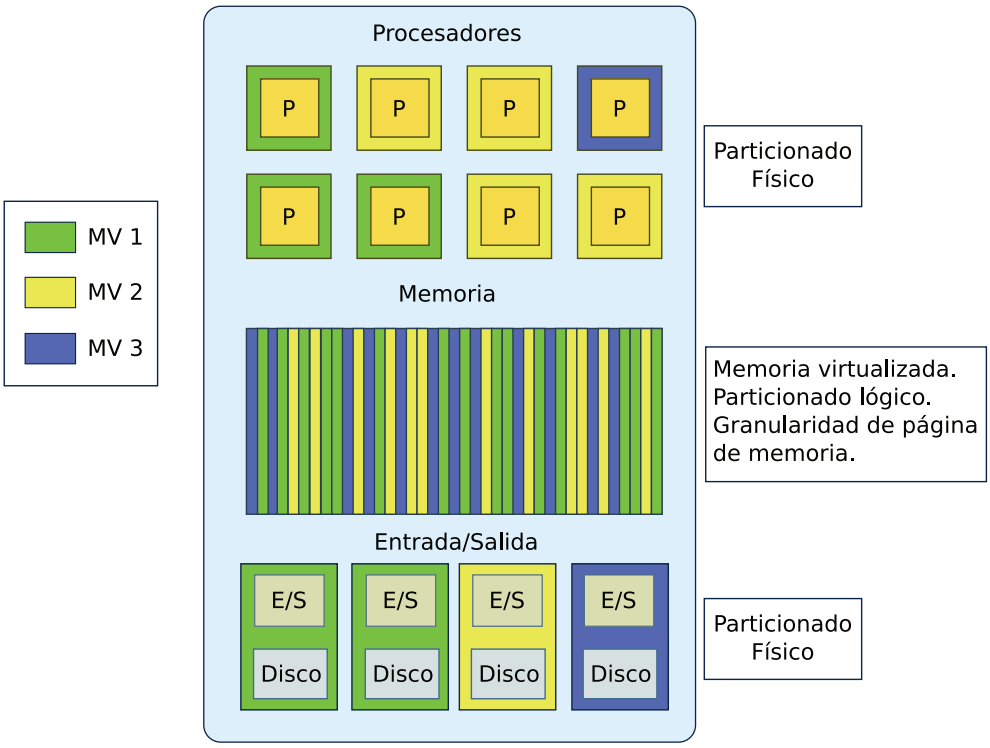

Figura 6.3: Nuestro esquema de virtualización. *Procesadores particionados físicamente. Memoria virtualizada a nivel de página de memoria. E/S particionada físicamente.*

mecanismos software. Sería un esquema muy similar al usado en el Cellular Disco, que nos permitiría llegar a implementar la compartición de páginas entre MVs.

En cuanto a la entrada/salida, realizamos particionado físico. Se reparten las unidades de E/S del sistema anfitrión entre las distintas MVs huéspedes.

El esquema que resume lo que acabamos de exponer puede observarse en la Figura 6.3.

Toda nuestra solución está basada en hardware (o firmware si es así como desea verse). La virtualización se encuentra implementada totalmente en el simulador.

Conforme vayamos construyendo nuestra infraestructura de simulación, explicaremos la forma en que se ha abordado la implementación de estos mecanismos de virtualización.

6.7. Estructura de Virtual-GEMS

Una vez tenemos resuelto el problema de separar GEMS de Simics, tal y como vimos en el Capítulo 5, deseamos desarrollar una primera aproximación a la virtualización haciendo uso de Ruby como simulador de sistema de memoria y Simics (u otro simulador) como simulador de un sistema completo.

Nuestra intención es la de conseguir unir varias instancias de Simics con una única instancia de Ruby. Con instancia nos referimos a un proceso. Es decir, cada instancia de Simics estará simulando un sistema completo independiente. Los sistemas simulados por distintas instancias de Simics pueden ser totalmente diferentes, en número de procesadores, en imagen de disco utilizada o en cualquier otra característica de sistema que podamos configurar con Simics. Nuestro interés, por tanto, está en ser capaces de conectar varias de estas instancias de Simics con una misma instancia de Ruby, de manera que todas los sistemas simulados usen el mismo subsistema de memoria. De esta manera estaríamos simulando la ejecución de máquinas virtuales, correspondiéndose cada instancia de Simics (y por tanto cada sistema simulado) con una máquina virtual. Obviamente para poder hacer esto hay que resolver antes varios problemas relacionados principalmente con el mapeo de las máquinas virtuales dentro del subsistema de memoria y con los problemas de concurrencia derivados de una infraestructura que ahora se convierte en un sistema distribuido.

Nuestra virtualización se lleva a cabo en el elemento RemoteDriver de RemoteRuby. Es el encargado de llevar a cabo la comunicación con las máquinas virtuales y gestionar las peticiones de memoria recibidas de cada máquina virtual para pasarlas al sistema de memoria simulado por Ruby. También se encarga de sincronizar a todos los elementos del sistema, esto es, todas las instancias de Simics (o el simulador que queramos usar) y el sistema de memoria. El esquema resultante se muestra en la Figura 6.4.

Una instancia de Simics proporciona a la máquina virtual los procesadores, así como los dispositivos de E/S. Así aseguramos el particionado físico que puede verse en la Figura 6.3. La virtualización de la memoria se lleva a cabo en RemoteRuby, que se encargará de llevar el control del reparto entre las máquinas virtuales de la memoria simulada en Ruby.

6.8. Comunicación

La comunicación desde el punto de vista de la máquina virtual es exactamente la misma que se describió en el caso de usar una única instancia de Simics. Puede verse en el Capítulo 5.5.

Inicialmente cada MV nos envía su configuración, consistente en la actualidad en tan sólo el número de procesadores que tiene la máquina virtual. RemoteRuby se encargará de mapear los procesadores de las máquinas virtuales de modo que se repartan entre los procesadores que aparecen en la topología simulada en Ruby. Usaremos varios mapeos diferentes, que se verán en el Capítulo 7.3. Lo importante ahora es saber que a cada procesador de cada máquina virtual le corresponde un procesador en Ruby.

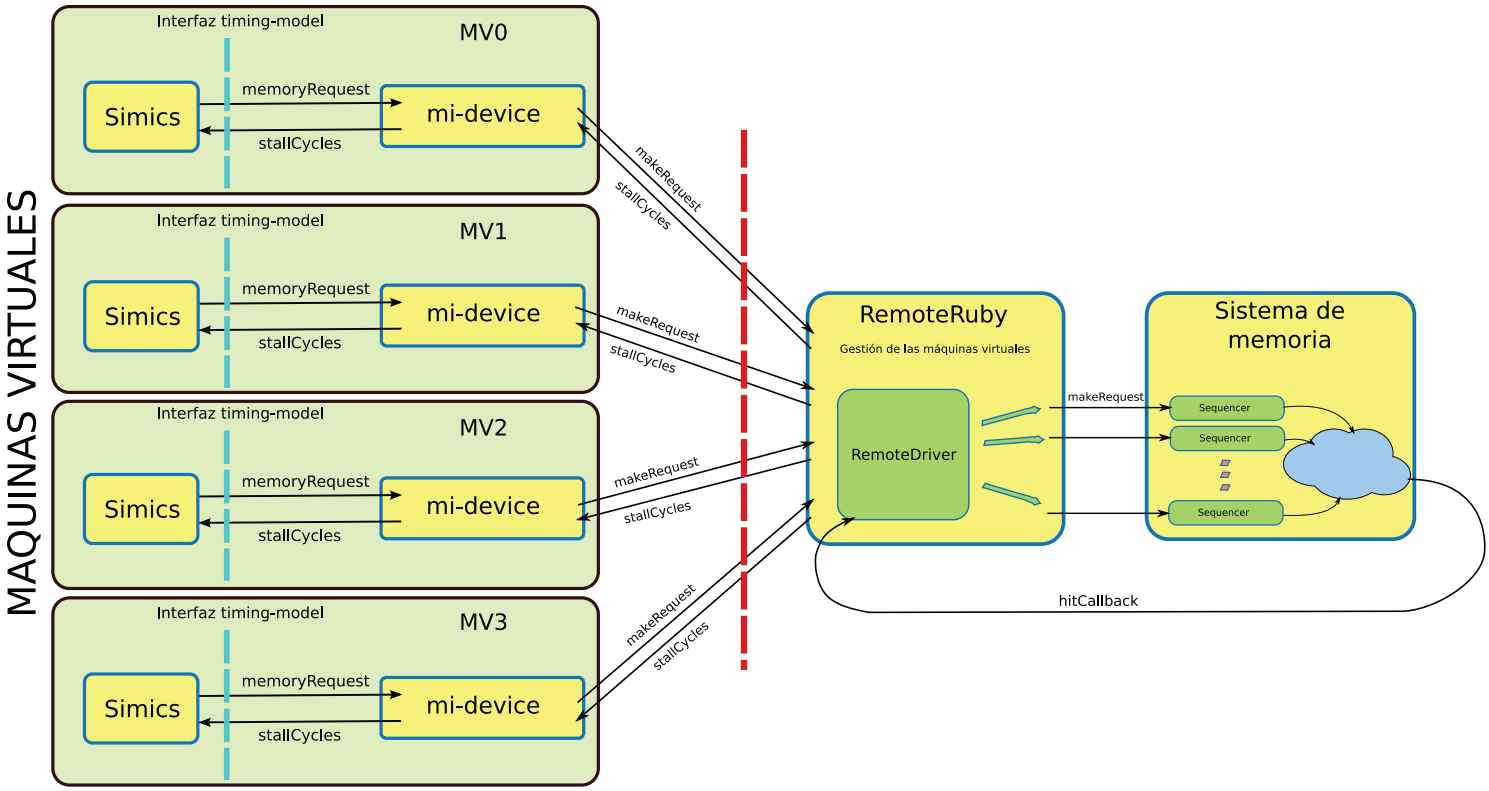

Figura 6.4: Esquema de Virtual-GEMS. *Podemos ver que cada máquina virtual se corresponde con una instancia de Simics. RemoteRuby, y concretamente RemoteDriver, es el encargado de la gestión de la virtualización.*

Desde el punto de vista del servidor, la situación se torna muy diferente. Ahora puede recibir peticiones de parte de varias instancias de Simics, lo que complica no la comunicación en sí, que sigue siendo la misma que antes con cada instancia de Simics concreta, sino el manejo concurrente de las conexiones de las MV.

Usaremos un hilo para servir las peticiones de cada máquina virtual conectada. El código de Ruby está pensado para la ejecución monohilo, por lo que sólo podemos tener en cada momento uno de los hilos que sirven peticiones ejecutando métodos de las clases que forman Ruby.

Ya hemos comentado varias veces cómo se sincronizan Ruby y Simics. La sincronización entre una instancia de Simics y Ruby es sencilla: se realiza a través de los mensajes triggerEvents (del cliente al servidor) y la respuesta triggerEventsACK (en sentido inverso), con los que se avanza un ciclo en Ruby. Sin embargo ahora tenemos varias instancias de Simics. La solución más sencilla al problema es implementar en el módulo RemoteRuby una barrera (a la que llamaremos "barrera triggerEvents") de manera que se avanzará un ciclo de Ruby cuando todas las instancias de Simics hayan realizado su triggerEvents correspondiente. En ese momento, y no antes, se lanzarán los eventos de la cola de Ruby correspondientes a ese ciclo, lo que puede provocar que se resuelvan accesos a memoria y se envíen por tanto mensajes *hitCallback* a las máquinas virtuales. Una vez se completó el disparo de los eventos, se responderá a todas las instancias con el triggerEventsACK de rigor, indicando que ya pueden seguir enviando las peticiones que correspondan al siguiente ciclo de Ruby. Precisamente de aquí surge una de las ventajas que aprovecharemos: las peticiones a memoria que se pasan a Ruby entre dos triggerEvents se considera que ocurren en el mismo ciclo de Ruby, por ello no importa el orden en que lleguen. De este modo, no tenemos que preocuparnos del orden que nos lleguen estas peticiones desde las máquinas virtuales entre la ejecución de dos barreras, ya que se considerará que todas ellas fueron lanzadas en el mismo ciclo. Tan sólo debemos preocuparnos de que las llamadas que hagamos a elementos de Ruby para pasar estas peticiones a los Sequencers se realicen en exclusión mutua.

6.8.1. Convención sobre triggerEvents

Para simplificar el código que debe controlar la comunicación, se llegó a la convención de que todos los mensajes que deban enviarse a una máquina virtual se enviarán cuando recibamos de ésta un triggerEvents, ya que entonces estará bloqueada a la espera del triggerEventsACK. Además, la barrera triggerEvents supone el único punto en el que podemos estar seguros de que, una vez todas las máquinas virtuales han enviado su triggerEvents, están todas bloqueadas sin excepción a la espera del triggerEventsACK, y por tanto podemos comunicarle a todas de manera "simultanea" cualquier situación que consideremos oportuno poner en su conocimiento. Una de estas situaciones es, por ejemplo, el final de la ejecución.

En el caso de una única instancia de Simics vimos que para terminar la ejecución, mi-device enviaba a RemoteRuby el mensaje de executionEnd, tras lo cual debía seguir ejecutándose normalmente, hasta que RemoteRuby respondiese con un executionEndACK. Esto está hecho así para permitir que las máquinas virtuales puedan terminar su ejecución en distintos instantes de tiempo. Cuando todas las máquinas hayan enviado la indicación de

que ha terminado su ejecución, en la siguiente barrera triggerEvents el servidor enviará a todas ellas el executionEndACK indicando que la ejecución de la simulación de todo el sistema ha concluido, y por tanto pueden terminar su ejecución.

En cada máquina virtual ejecutaremos una carga de trabajo (concretamente las cargas de la suite SPLASH-2 que ya tenemos preparadas), y cada máquina virtual acabará su ejecución en un momento distinto. Debido a que estaremos interesados en obtener estadísticas del momento final de la ejecución de la carga que más tarde en ejecutarse, no nos interesa que cuando otras máquinas virtuales terminen previamente con la ejecución de sus respectivas cargas, dejen de realizar peticiones de memoria. Por esta razón, el proceso que se sigue para la finalización de la simulación es el que se ha comentado, en que todas las MVs continúan activas hasta que la última termine, ya que así todas las máquinas virtuales estarán trabajando (y por tanto influyendo en el rendimiento) hasta el momento en que queramos obtener las estadísticas. Si no fuese así cada vez habrían menos peticiones a memoria hasta llegar a un momento en que sólo la última máquina virtual realizase accesos. En la sección 6.10.2 puede encontrarse una discusión más extensa sobre esta cuestión.

6.9. Estadísticas

Otro tema que debemos tratar es el de las estadísticas de Ruby. En el caso de una única instancia de Simics, veíamos como el envío de un mensaje clearStats al servidor provocaba la limpieza de las estadísticas, mientras que un dumpStats provocaba que se obtuvieran las estadísticas de la ejecución hasta ese ciclo.

En el caso de usar varias máquinas virtuales existen varias opciones. La elegida consiste en que cuando cualquier máquina virtual ejecuta un clearStats, las estadísticas se limpian. Normalmente una MV realizará su calentamiento (principalmente para llenar las cachés), tras ello ejecutará un clearStats, para posteriormente comenzar la ejecución en sí de la carga de trabajo, de la que deseamos obtener estadísticas. Con la aproximación que hemos tomado, será cuando la última MV termine su calentamiento cuando se produzca la última limpieza y se comience a tomar estadísticas.

Para el caso de la recolección de las estadísticas, normalmente, las cargas están preparadas para ejecutar un número determinado de transacciones (esto es, una cantidad de trabajo concreta) y tomar estadísticas al final de la ejecución. Por ello, cuando la última máquina virtual indique que desea tomar las estadísticas, será el momento en que éstas se vuelquen, de manera que las estadísticas sean resultado de *todo* el trabajo de *todas* las máquinas virtuales.

6.10. Cargas usadas

Un benchmark se define como un estándar de medida o evaluación. Un benchmark en el ámbito de los computadores es típicamente un programa de ordenador que realiza un conjunto de operaciones estrictamente definido (una carga de trabajo o *workload*) y devuelve alguna forma de resultado (una métrica) describiendo como se comportó la máquina

probada. Las métricas de los benchmarks normalmente miden la velocidad (como de rápido se completó la carga de trabajo) o la productividad (cuantas cargas de trabajo por unidad de tiempo se midieron). Ejecutar el mismo benchmark en distintos ordenadores permite hacer una comparación.

Al igual que en las máquinas reales, para realizar las medidas de rendimiento de una máquina simulada se usan cargas de trabajo que son ejecutadas dentro de la simulación. Estas cargas pueden ser de diversa naturaleza.

Existen benchmarks estandarizados, que pueden usarse como punto de referencia. En nuestro caso utilizaremos los benchmarks de la suite SPLASH-2. Éstos se centran en el rendimiento de la máquina en cómputo intensivo, lo que significa que enfatizan el rendimiento del procesador o procesadores de la máquina y la arquitectura de memoria.

Una de las razones por las que esta suite de benchmarks se adapta bien a nuestras necesidades es que no estresa otros elementos del sistema, como I/O (unidades de disco), redes o gráficos. SPLASH-2 [30] cumple con estos requisitos, ya que se centra en el rendimiento de la CPU/sistema de memoria.

6.10.1. Preparación de las cargas

Simics permite la obtención de *checkpoints*, esto es, instantáneas del estado de la máquina. Para ejecutar un *workload*, lo primero que se hace es obtener una instantánea de la máquina en el instante justo en el que su ejecución va a dar comienzo, para poder así ejecutarlo tantas veces como queramos sin necesidad de volver a repetir ninguno de los pasos anteriores (instalación del benchmark, arranque del sistema, etcétera).

Sin embargo, el estado del sistema de memoria (esto es, de Ruby), no se almacena en el *checkpoint*. Por tanto, cuando carguemos el *checkpoint* el sistema de memoria se encuentra sin inicializar (las cachés estarán vacías). Antes de comenzar a tomar medidas, necesitaremos calentar las cachés para que el llenado de las mismas no influya en la medida del rendimiento.

Los *workloads* se utilizan ejecutando *transacciones*. Para nuestro propósito, consideramos una transacción como una unidad de trabajo realizado, que puede consistir desde una iteración del algoritmo que se ejecuta en el benchmark, hasta una ejecución completa del benchmark.

Para calentar la caché se ejecutará un numero determinado de transacciones del *workload*. Tras ellas comenzarán a ejecutarse las transacciones correspondientes al periodo de medida de rendimiento. Una vez se complete el número de transacciones estipulado para este periodo, pasaremos a la fase de espera del fin de la ejecución de todas las MVs.

El esquema de la ejecución de un *workload* en una MV quedaría tal y como puede verse en la Figura 6.5.

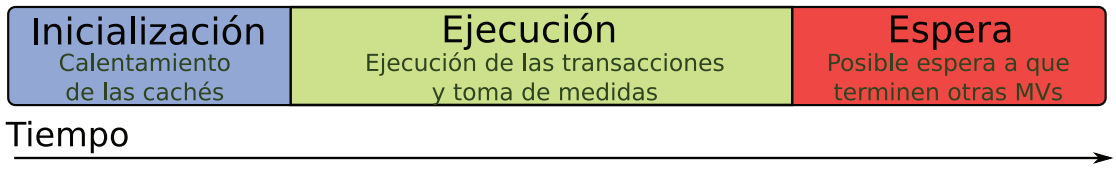

Figura 6.5: Fases en la ejecución de un *workload*.

6.10.2. Problemas con las cargas

Como hemos visto, la ejecución de un *workload* en una MV se divide en tres fases: inicialización, fase de medición y espera.

Cada máquina virtual que usemos en el sistema puede usar un *workload* distinto. La ejecución de cada *workload* tendrá una duración distinta en sus fases, si bien la ejecución total tendrá la misma duración en todos los *workloads* de las MVs, ya que todas las MVs comienzan su ejecución en el mismo instante, y todas finalizan cuando haya terminado la fase de medición de la última de ellas.

La situación que se nos platea queda descrita en la Figura 6.6.

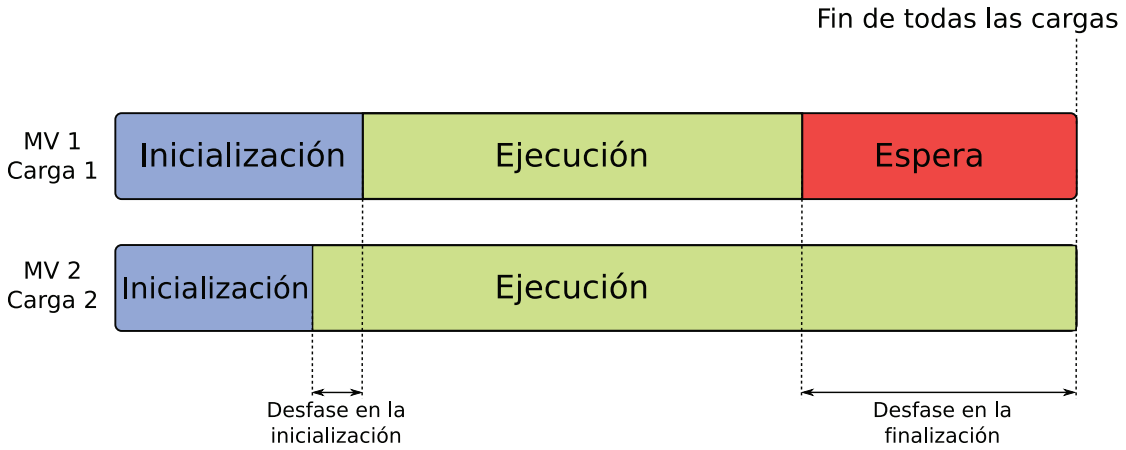

Figura 6.6: Fases de ejecución que atraviesan los *workloads* de las máquinas virtuales. *Obsérvense los desfases que aparecen entre distintas cargas de trabajo.*

Como vemos, puede haber *workloads* que terminen antes su inicialización, y también que terminen antes su fase de medición, teniendo que esperar a los que más tiempo tomen en ellas.

Esto plantea varios interrogantes a la hora de la obtención de resultados de rendimiento. Son los siguientes:

¿Cuándo comienza la toma de estadísticas? Si comenzamos a tomar estadísticas cuando la primera máquina virtual indique que comienza a tomar medidas, tendremos un periodo de tiempo en el que estaremos midiendo el rendimiento mientras algunas máquinas virtuales aún están realizando el calentamiento, con lo que estaríamos viciando los resultados. Esto puede verse en la Figura 6.6: estaríamos tomando medidas

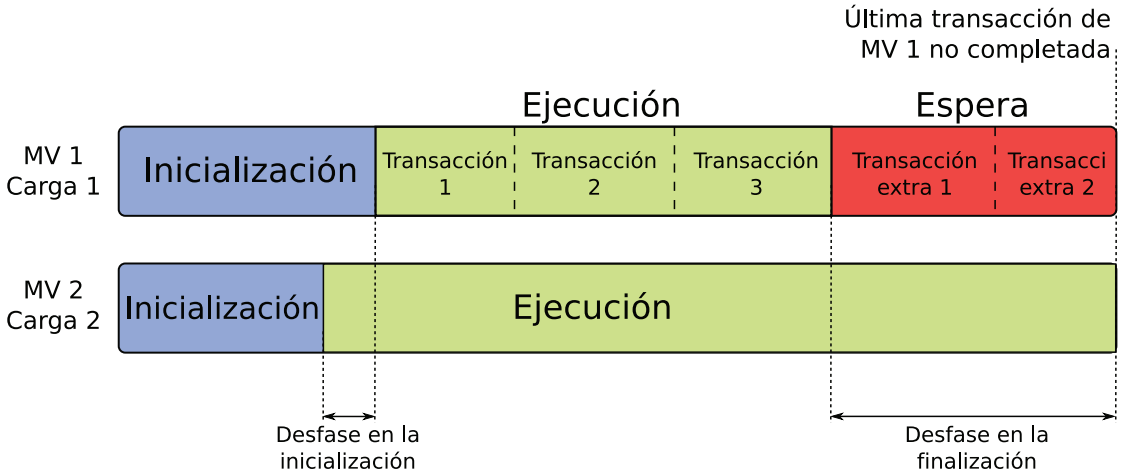

Figura 6.7: Transacciones Extra. *La MV 1 ejecuta durante su fase de espera transacciones hasta que termine la ejecución del resto de MVs.*

durante el desfase en la inicialización, mientras la MV 1 aún no está realizando trabajo útil. Si por el contrario comenzamos a tomarlas cuando la última MV haya terminado de calentar, habremos perdido la medición de parte del trabajo de las MVs que terminaron antes el calentamiento. En la Figura 6.6 podemos apreciar que perderíamos el trabajo que la MV 2 realiza durante el desfase en la inicialización.

¿Cuando terminan de tomarse estadísticas? La situación es la misma que en el caso de comenzar a tomarlas. Como puede verse en la Figura 6.6, al no terminar la ejecución de todas las máquinas virtuales de manera simultanea, tenemos el problema de que, o estamos tomando medidas durante el desfase en la finalización, tras el fin la ejecución de la MV 1, o estamos dejando de tener en cuenta trabajo realizado por la MV 2 durante el desfase en la finalización.

Lo ideal sería conseguir unos *workloads*, de los que no disponemos actualmente, en los que pudiéramos conseguir que el tiempo de calentamiento fuera aproximadamente igual, y el tiempo de ejecución también. Una opción para ello es ajustar el número de transacciones que se ejecutan en cada *workload* para el calentamiento y para la ejecución, de manera que los tiempos necesarios por las distintas fases fueran aproximadamente los mismos en todas las MVs.

Sin embargo, la heterogeneidad de lo que se consideran transacciones entre los distintos *workloads* hacen que este ajuste sea complicado, ya que hay casos en que puede llegar a considerarse como una transacción la ejecución completa del algoritmo, dificultando por tanto el ajuste del tiempo de ejecución de los *workloads*.

Otra opción, quizá la más atractiva, es hacer que, tras el término del periodo de medición de un *workload*, se ejecuten nuevas transacciones hasta que todas las MV hayan terminado su periodo de ejecución. De ese modo, la MV seguiría realizando trabajo útil, el mismo que realizaba durante su etapa de medición, pero ahora la última transacción no se completará al verse interrumpida por el término de la ejecución de la simulación, como puede verse en la Figura 6.7. Con esto conseguiríamos solucionar el problema de cuándo terminar de tomar estadísticas.

En nuestro caso, la solución adoptada es comenzar a tomar estadísticas cuando la última MV termine su fase de inicialización, y terminar de tomarlas cuando la última MV acabe su ejecución. Para solventar el problema del desfase en la finalización, en las pruebas usaremos el mismo tipo de *workload* en todas las MVs de una simulación (si bien la infraestructura permite combinar en una simulación los *workloads* que queramos). De esta manera nos situamos en el caso deseado en que las fases de inicialización y de toma de medidas de las distintas MVs tienen aproximadamente la misma duración.

7

Validación de Virtual-GEMS

Debemos comprobar el correcto funcionamiento de nuestra infraestructura de virtualización. También obtendremos unos primeros resultados de cómo varía el rendimiento del sistema simulado según los parámetros de diseño usados. Concretamente, veremos diversos mapeos de memoria que se han implementado en Virtual-GEMS.

7.1. Arquitecturas simuladas

Simularemos dos arquitecturas de multiprocesador. La arquitectura del sistema de memoria y la ubicación de los procesadores queda definida en Ruby, ya que es allí donde se establece la topología de la red que une los procesadores y los distintos elementos de la jerarquía de memoria, así como las características de latencia, ancho de banda y tamaños de esos elementos. Por tanto, de Simics sólo nos interesa saber cuantos procesadores forman cada máquina virtual, Ruby se encargará de colocar dichos procesadores dentro de la topología de la red.

En cuanto a la frecuencia de funcionamiento, consideraremos de momento que todas las máquinas simuladas por Simics funcionan con un reloj principal a la misma frecuencia. Ruby se basará en dicho reloj gracias a la ya explicada sincronización que existe entre las máquinas virtuales y el sistema de memoria simulado por Ruby.

Para empezar tendremos un *tiled* CMP (Chip MultiProcessor) [21] de 16 procesadores. Este está formado por un único chip donde se encuentran los 16 *tiles*, conteniendo cada *tile* un procesador, su caché L1 y una parte de la caché L2 del chip, siendo esta última compartida entre los *tiles*. La memoria RAM es centralizada y los procesadores se comunican con ella a través del controlador de memoria, que está a igual distancia de todos ellos. Su estructura puede observarse en la Figura 7.1.

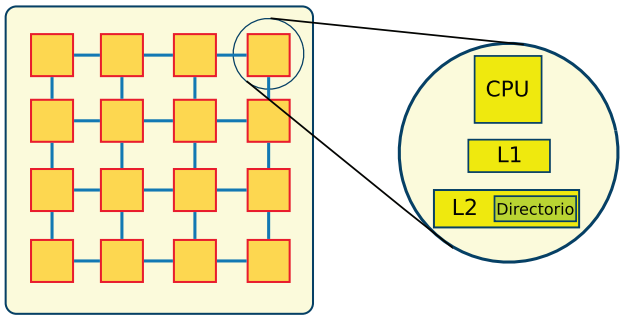

Figura 7.1: Estructura de un *tiled* chip-multiprocessor con 16 procesadores.

Un *tile* está formado por procesador, caché L1 privada (dividida en caché de datos e instrucciones) y una porción de la caché L2 compartida (que contiene el directorio L2). La caché L2 de todo el chip es compartida entre los procesadores. Es decir, lógicamente el chip sólo tiene una caché L2, y todos los procesadores ven la jerarquía de memoria a través de la misma L2, pero físicamente la L2 se reparte entre los distintos *tiles* del chip. Por ello los problemas de coherencia aparecen a nivel de L1 y necesitamos el directorio en L2 para poder mantener dicha coherencia.

Al usar una L2 compartida, cada *tile* tiene un "trozo" o *slice* de dicho nivel de caché. La estrategia que se sigue para asignar los bloques de caché a los *slices* es usar varios bits de la dirección física accedida para mapear el bloque a un *slice* concreto. Es decir, dada una dirección de memoria sabemos a qué *slice* irá el bloque que contenga a esa posición de memoria. Un *slice* de L2 se corresponde físicamente con un banco de L2. En adelante nos referiremos a la L2 de cada procesador como su banco de L2. En la Figura 7.2 puede verse como se realiza esta asignación de la memoria a la caché.

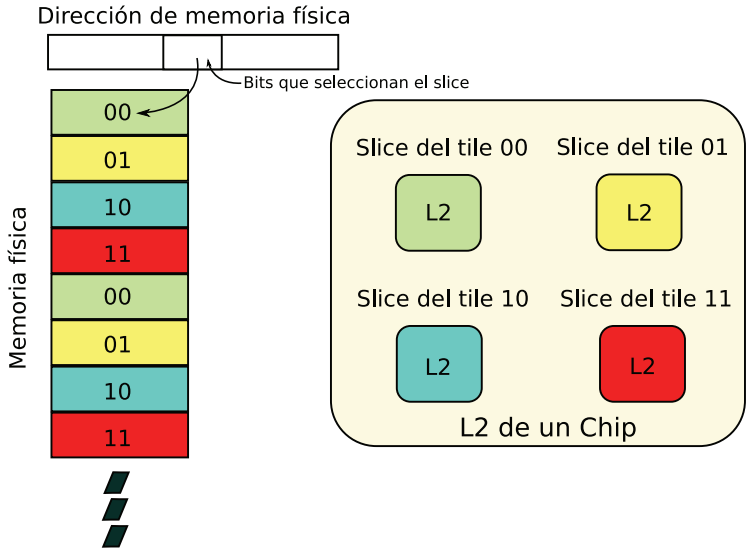

Figura 7.2: Mapeo de la memoria en los *slices* de L2 de un chip. *En la dirección de memoria física hay un conjunto de bits que se usan para elegir el slice de L2 al que irá el bloque. Dependiendo de qué bits de la dirección se usen para dicho mapeo, los trozos contiguos de memoria que corresponden a los slices serán más o menos grandes.*

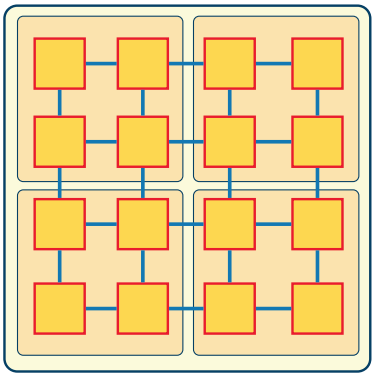

Figura 7.3: Estructura de un multiple chip-multiprocessor con 4 chips de 4 procesadores cada uno.

La segunda arquitectura que usaremos es un Multiple-CMP (M-CMP, Multiple-Chip MultiProcessor), Figura 7.3. En este caso tenemos cuatro chips, conteniendo cada uno de ellos cuatro *tiles*, cada *tile* con un procesador, su L1 y una parte de la L2 del chip. Cada chip tiene su propia L2, pero dentro de un chip la L2 se comparte entre los procesadores de la misma forma que se explicó para la arquitectura *tiled* CMP.

Hemos definido esta arquitectura de modo que las latencias entre los elementos sea exactamente la misma que en el caso del *tiled* CMP. En este caso, igual que en el del *tiled* CMP, el acceso a memoria se hace mediante un controlador que está a igual distancia de todos los procesadores. Además, el tamaño de la caché L1 y de los bancos de L2 es el mismo que en el caso del *tiled* CMP.

La única diferencia con respecto a la arquitectura *tiled* CMP es que, al tener cada chip del M-CMP una caché L2, ahora tenemos cuatro cachés L2, mientras que antes, en el *tiled* CMP, teníamos tan sólo una L2 compartida por todos los procesadores. Como el tamaño de los bancos es el mismo, cada una de las cuatro cachés L2 del M-CMP tiene un tamaño que es la cuarta parte de la L2 del *tiled* CMP. Cada caché L2 del M-CMP es compartida por los cuatro procesadores del chip que contiene esa caché.

El resto de características del *tiled* CMP y del M-CMP son las mismas, incluídas todas las latencias de la red de interconexión. Por tanto, no estaríamos ante un verdadero M-CMP, sino que así conseguimos modelar un *tiled* CMP con varias cachés L2 [15].

7.2. Máquinas virtuales

Es posible usar el número de máquinas virtuales (MVs) que deseemos, y cada máquina virtual puede tener el número de procesadores que necesitemos. En nuestro caso usaremos cuatro máquinas virtuales, cada una de ella con cuatro procesadores, para tener así los 16 procesadores que forman cada una de las arquitecturas que usaremos.

Cuando una máquina virtual realiza una acceso a memoria, el procesador realiza el acceso a una dirección virtual. Vimos en la sección 6.4.2 que estábamos simulando dos niveles de tablas de páginas. El primero sirve para traducir el número de página virtual a

número de página real (que es la página que la MV cree como física, pero que en realidad no lo es). El segundo nivel hace la traducción de página real de una MV a página física (con la que ya se puede acceder a la memoria física). Vimos los métodos que había para que se realizasen estas dos traducciones de manera transparente para la MV.

Llamaremos **direcciones virtuales** a las direcciones que ven los procesadores de las máquinas virtuales. Una **dirección global** será un par (máquina virtual, dirección real) que identifica unívocamente una dirección dentro del sistema.

Una dirección virtual accedida por un procesador de una máquina virtual se traducirá a una dirección global mediante la traducción realizada por la MMU indicada anteriormente y añadiendo el identificador de la máquina virtual que realiza la petición.

A partir de la dirección global habrá que hacer una nueva traducción a una **dirección física** dentro del sistema de memoria simulado por Ruby.

En nuestra implementación, la primera traducción se lleva a cabo en Simics, mientras que la segunda se hace en el RemoteDriver cuando recibe una petición de memoria. El efecto es el mismo que el que se obtiene con las técnicas usadas realmente y explicadas en la sección 6.4.2.

7.3. Mapeo de los procesadores

Es hora de ver el mapeo de los procesadores de las máquinas virtuales (es decir, los procesadores de cada instancia de Simics) a los procesadores dentro de la arquitectura simulada en Ruby.

En el caso de la arquitectura *tiled* CMP, el mapeo puede observarse en la siguiente Figura 7.4.

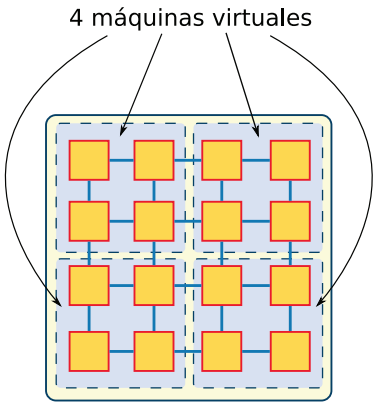

Figura 7.4: Mapeo de los procesadores de las MVs en el *tiled* CMP. Cada MV usa 4 procesadores.

Cada máquina virtual abarcará cuatro procesadores de manera que la latencia entre ellos sea la mínima posible (de ahí la forma en cuadrado que adopta la máquina virtual dentro de la arquitectura).

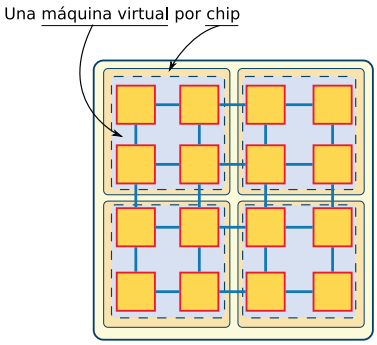

Figura 7.5: Mapeo de los procesadores de las MVs en el M-CMP. Cada MV usa los procesadores de un chip.

En el caso del M-CMP, cada máquina virtual se corresponderá con un chip, siendo la disposición de éstas prácticamente la misma que en el caso del *tiled* CMP. Podemos verlo en la Figura 7.5.

La principal diferencia que se deduce de ambas configuraciones es que en el caso del *tiled* CMP las MVs comparten la caché L2, mientras que en el caso del M-CMP cada MV tiene su propia caché L2. Realmente, el único motivo del uso de un M-CMP es conseguir que cada MV tenga su propia caché. Como dijimos anteriormente, las latencias de la red de interconexión son las mismas en el *tiled* CMP y en el M-CMP, por lo que el hecho de que haya uno o cuatro chips no significa nada más allá de esta diferencia en la forma de repartir la L2.

7.4. Mapeos de memoria

Existen diversas posibilidades a la hora de realizar el mapeo de las posiciones de memoria de las maquinas virtuales (direcciones globales) en el espacio del sistema de memoria simulado por Ruby (direcciones reales). La forma en que se haga afectará al mapeo de los bloques de memoria a los bancos de L2 [7], afectando también al rendimiento, como veremos.

En nuestro caso hemos optado por realizar el mapeo a nivel de página de memoria. El tamaño de página que consideramos es el que usa Ruby, definido mediante la variable de configuración g_PAGE_SIZE_BYTES. Por tanto cada página de cada máquina virtual se mapeará a una página del sistema de memoria global.

Consideraremos que una página global está formada por el par (máquina virtual, número de página real). Cada página global se traduce a una página física en el espacio de memoria de Ruby.

En estos momentos no contemplamos la compartición de páginas entre las máquinas virtuales, por lo que cada página global irá a un lugar distinto en la memoria física de Ruby.

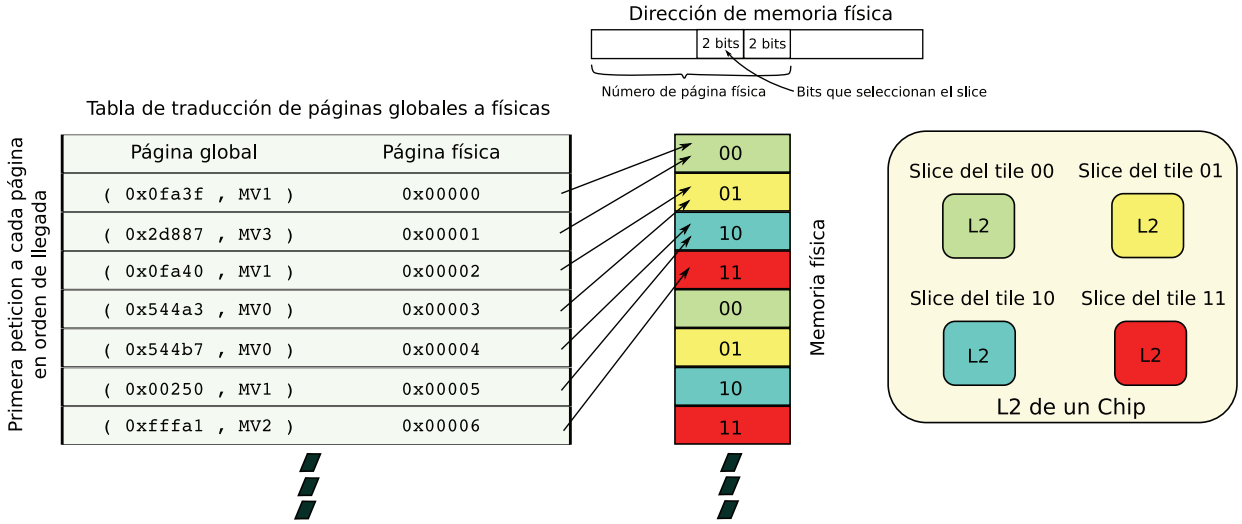

Figura 7.6: Mapeo de memoria real a física simple. *Podemos ver la tabla que se va creando conforme llegan peticiones a páginas globales que no tienen aún mapeo a memoria física. A cada nueva página global se le asigna la siguiente página física libre. Como las peticiones a páginas globales pueden venir de cualquier MV, no podemos predecir a qué slice se mapearán las páginas a las que accede cada MV.*

El primer mapeo, al que llamaremos mapeo simple, consiste en asignar a cada página global la siguiente página física disponible. Comenzaremos por la página cero, y conforme lleguen peticiones de memoria a posiciones de páginas globales que aún no han sido mapeadas les iremos asignando la siguiente página física libre. Esta forma de asignar las páginas puede verse en la Figura 7.6.

Se trata del mapeo más sencillo. En el fondo es muy similar a repartir las páginas de las máquinas virtuales aleatoriamente por la memoria física. La memoria física es centralizada y las páginas físicas no están más cerca de unos procesadores que de otros, por lo que la posición de una página en memoria no afecta a la latencia de los accesos. Lo que afecta al rendimiento es la distancia del procesador al banco al que se mapea la página a la que quiere acceder. En este caso, al ser mapeo aleatorio en memoria, el banco donde se mapeará cada página será aleatorio. Este mapeo lo usaremos tanto con la arquitectura *tiled* CMP como con la arquitectura M-CMP.

El siguiente mapeo que hemos implementado, basándonos en el trabajo de Sangyeun Cho et al. [7], es al que llamaremos **Virtual Machine Cache Bank Mapping (VMCBM)**. Su funcionamiento es bastante sencillo. Se dan tres circunstancias sobre las que basamos este mapeo:

- No existe la compartición de páginas entre las máquinas virtuales.
- A partir de una dirección de memoria se deduce en qué *slice* de L2 se cargará el bloque al que pertenece. Es decir, podemos saber en la caché de qué procesador del chip debe encontrarse dicho bloque.
- Los bits de una dirección de memoria que se usan para determinar dicho *slice* se encuentran dentro del número de página. Esto implica que cualquier dato de una

misma página de memoria física se cargará siempre en el mismo banco de L2. Esto nos permite que al hacer la MMU la traducción de número de página global a número de página física podamos predecir a qué banco irán a parar los datos de dicha página global que sean accedidos. Es decir, se produce un mapeo directo de página de memoria a banco de L2.

Usaremos 4 GB de RAM en el sistema, por lo que 32 bits serían suficientes para direccionar toda la memoria. En la Figura 7.7 vemos como se usan los bits menos significativos para hacer corresponder la dirección de memoria con un lugar físico en concreto dentro de las cachés L2.

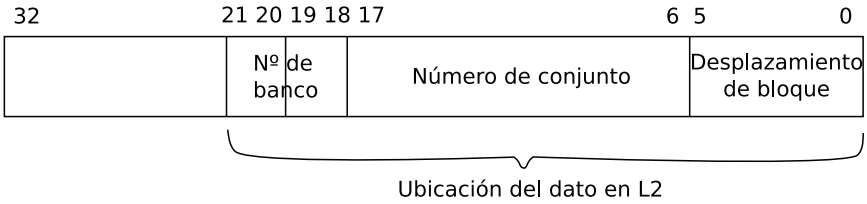

Figura 7.7: Uso de los distintos rangos de bits de una dirección de memoria. *Los bits 0-5 se usan para seleccionar el byte dentro del bloque de caché (bloques de 64 bytes). Los bits 6-17 para seleccionar el conjunto dentro del banco de L2 en que se encuentra el bloque (512 conjuntos por banco). Los bits 18-21 en el caso de usar un chip de 16 procesadores (tiled CMP), o 18-19 en el caso de chips de 4 procesadores (M-CMP), se usan para seleccionar el banco del L2.*

Gracias a todo esto, podemos realizar el mapeo de forma que cuando una máquina virtual acceda a un dato de memoria, la página global a la que pertenece dicho dato se asigne a una página física cuyos datos se lleven cuando sean accedidos a la caché un procesador de esa máquina virtual y no de otra. Para conseguirlo, tan sólo hay que fijar los bits de la dirección que determinan a qué *slice* le corresponden los datos, para que ese *slice* pertenezca a un procesador de la máquina virtual. Puede verse este mapeo en la Figura 7.8.

Cada vez que tengamos una nueva página global que mapear, iremos mapeándola a la L2 de un procesador distinto de la máquina virtual que hace la petición, siguiendo una estrategia round robin.

Con esto aprovechamos la no compartición de páginas entre las máquinas virtuales haciendo que los datos accedidos por una máquina virtual siempre haya que buscarlos en la caché de los procesadores de dicha máquina virtual, y nunca en los de otra, como sucedía en el protocolo simple. Así evitamos tener que recorrer más enlaces del chip para buscar los datos en L2 cuando haya fallos de L1.

Usaremos el mapeo VMCBM con la arquitectura *tiled* CMP. No lo usaremos con M-CMP porque en esta arquitectura ya se mapean los bloques de memoria usados por una MV a la caché del chip donde se encuentra dicha MV. El *tiled* CMP con este mapeo y el M-CMP con el mapeo simple son muy similares. La única diferencia apreciable es que en el *tiled* CMP con mapeo VMCBM las páginas siempre se repartirán equitativamente entre los *slices* L2 de las máquinas virtuales debido a nuestra estrategia round robin al mapear; mientras que en el M-CMP este reparto no tiene por qué ser equitativo, ya que depende de

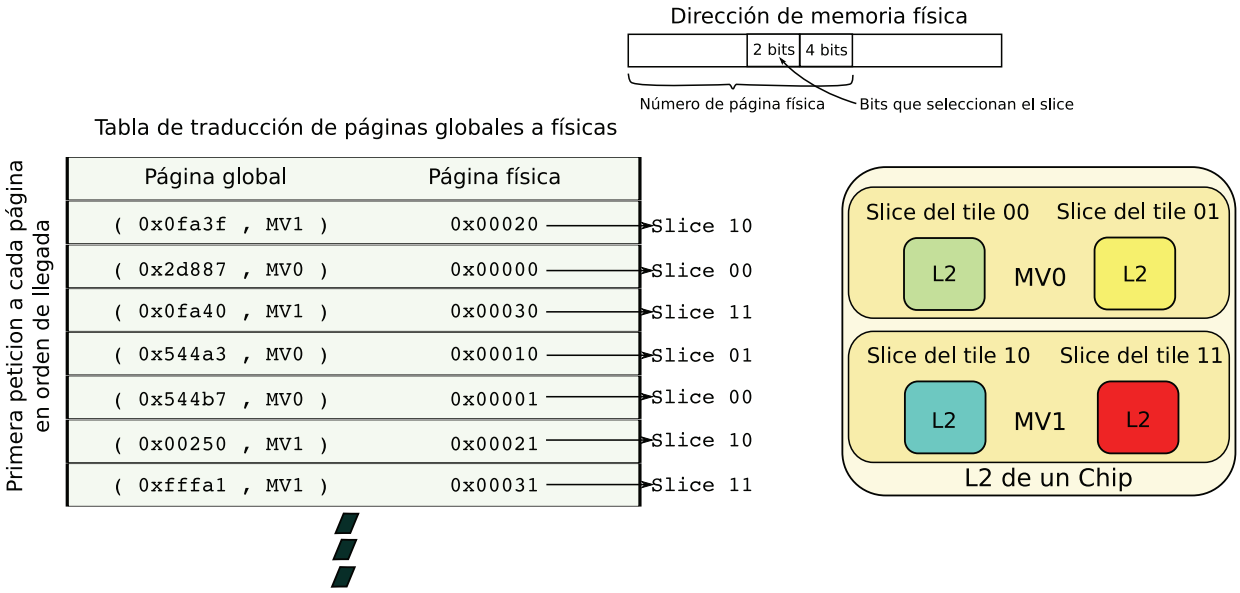

Figura 7.8: Mapeo de memoria real a física VMCBM. *Puede verse que los slices 00 y 01 de L2 le corresponden a la MV0 y los slices 10 y 11 a la MV1. Si nos fijamos en la tabla de traducción, vemos como a cada página global de una MV le asignamos una página que se mapea a un slice perteneciente a dicha MV. Además vamos rotando los slices de una MV virtual, usando round-robin, cuando llegan nuevas páginas globales que mapear. Así se reparten equitativamente las páginas accedidas por una MV entre sus slices.*

las páginas físicas que vayan asignándose a cada máquina virtual al hacer el mapeo simple. Puede verse la diferencia en la Figura 7.9.

El siguiente mapeo que hemos desarrollado es al que llamamos **Tile Cache Bank Mapping (TCBM)**. Al igual que el anterior mapeo, nos basamos en la no compartición de páginas entre MVs y la posibilidad de mapear directamente una página de memoria a un banco de L2 concreto. En este caso, en lugar de mapear las páginas globales accedidas por una máquina virtual a la L2 de los procesadores de dicha máquina virtual, lo que haremos será un mapeo aún más fino, llevando las páginas accedidas por un procesador concreto de la máquina virtual al banco de L2 de dicho procesador.

Antes, con el mapeo VMCBM, las páginas accedidas por un procesador podían estar en el banco de L2 de cualquier procesador de la máquina virtual. Ahora intentamos hacer que siempre que sea posible estén en el procesador que las accede. A diferencia de lo que sucedía entre las máquinas virtuales, donde no había compartición, dentro de una máquina virtual, entre los procesadores de ésta, sí que existe compartición, y es probable que una página accedida por un procesador de la MV también sea accedida por otro. Sin embargo, de media los datos accedidos por los procesadores estarán en un *slice* más cercano que usando VMCBM. En el peor caso estarían igual de lejos. Pero debido a que las aplicaciones paralelas (sobre todo las de cálculo científico) suelen repartir el trabajo a realizar entre los distintos procesadores (repartiendo también los datos sobre los que realizar los cálculos) se reduce la compartición.

Pero este mapeo, TCBM, tiene un problema. Si un procesador accede antes que los otros a la mayoría de los datos usados (por ejemplo para inicializarlos), o simplemente realiza

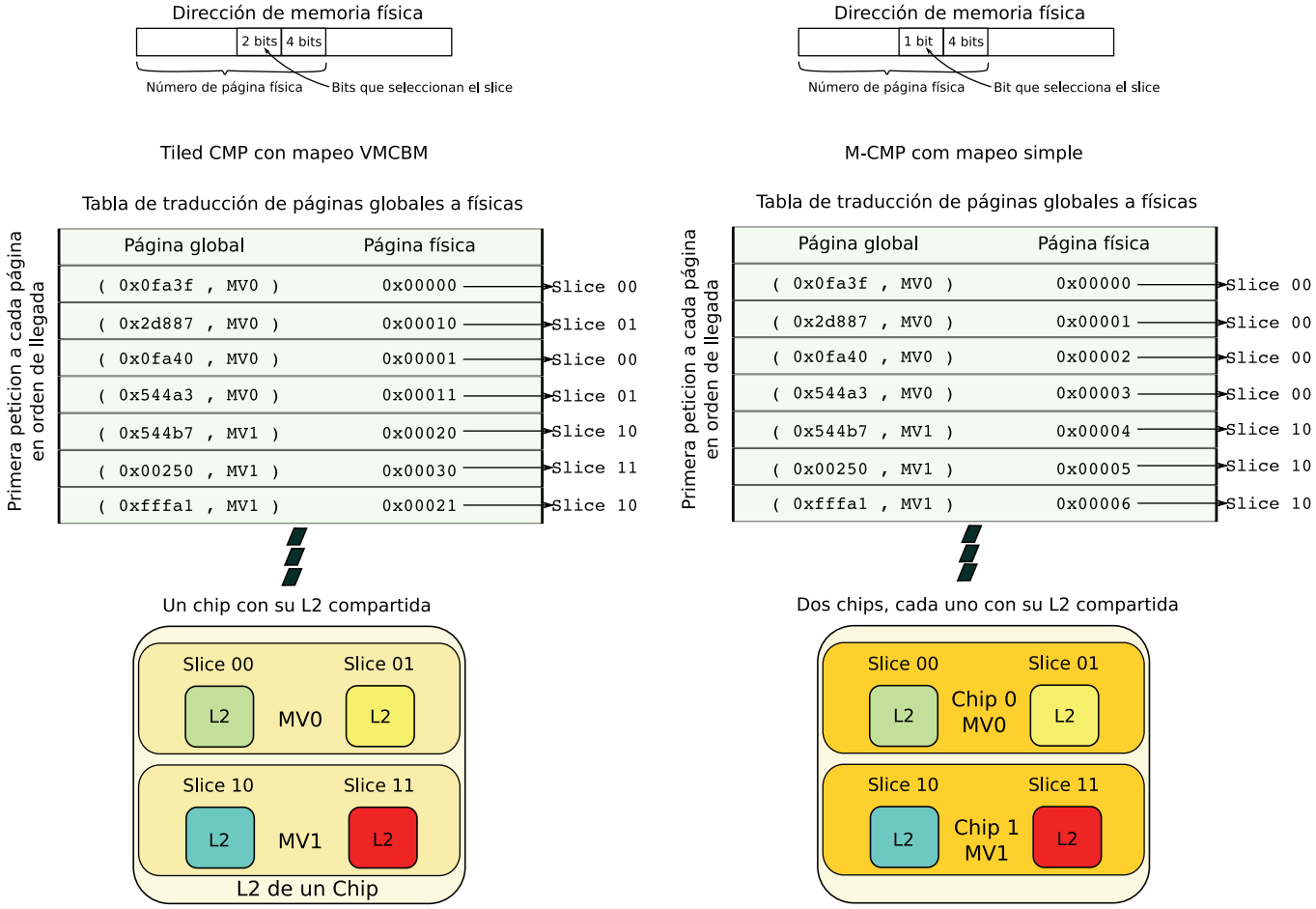

Figura 7.9: Diferencia entre *tiled* CMP con mapeo VMCBM y M-CMP con mapeo simple. *En el caso del tiled CMP se usan dos bits de la dirección para realizar el mapeo, ya que el chip tiene cuatro tiles. Por el contrario, en el M-CMP se usa un solo bit, ya que los bloques de cada MV se mapean a los slices del chip que ocupa dicha MV, y cada chip tiene dos slices. Como vemos, el mapeo en el caso del CMP con VMCBM es equitativo entre los tiles de una MV, al aplicar estrategia round robin. En el caso del M-CMP con mapeo simple no sucede eso. En este ejemplo vemos como todos las páginas accedidas por la MV0 van a parar a un mismo slice del chip en que se encuentra la MV, y a las accedidas por la MV1 les sucede lo mismo.*

trabajo no paralelizable con datos a los que sólo él accede, se producirá una situación de desequilibrio en el reparto de las páginas de memoria entre los bancos de L2, y habrá una "sobrecarga" de datos en el *slice* de este procesador, produciéndose a su vez un desaprovechamiento de los otros *slices* y elevándose la tasa de fallos en L2. Se podrían tomar medidas para detectar estas situaciones e intentar paliarlas, complicando el mapeo, pero en principio usaremos el mapeo TCBM puro.

Usaremos el mapeo TCBM con la arquitectura *tiled* CMP. En realidad da igual usarlo en el *tiled* CMP que en el M-CMP, el resultado será muy similar, ya que en ambos casos tendríamos que las páginas accedidas por cada procesador irían a su banco, independientemente de que la L2 sea compartida por los 16 procesadores (*tiled* CMP) o sea compartida por cada cuatro procesadores (M-CMP).

En total tenemos cuatro máquinas distintas con las que realizar las pruebas:

- Arquitectura *tiled* CMP con mapeo simple. Las máquinas virtuales comparten la L2.
- Arquitectura M-CMP con mapeo simple. Mejora con respecto a la anterior en que cada máquina virtual tiene su propia L2.
- Arquitectura *tiled* CMP con mapeo VMCBM. Mejora con respecto a la anterior en que ahora mapeamos las páginas en orden round-robin entre las *slices* de la máquina virtual.
- Arquitectura *tiled* CMP com mapeo TCBM.

Los mapeos, tanto el de procesadores de las máquinas virtuales a procesadores en Ruby como el de páginas globales a páginas físicas, se realizan en el Driver (que en este caso estaría actuando como Hypervisor de las máquinas virtuales), y el código está organizado de tal manera que sea posible añadir nuevos mapeos fácilmente. Por ejemplo, para añadir un nuevo mapeo de procesador tan sólo hay que añadir un par de matrices que lo describan.

En estos momentos no consideramos tampoco el reemplazo de páginas. De manera que la memoria de Ruby debe ser lo suficientemente grande como para contener todas las páginas globales mapeadas a páginas físicas. Las cargas de trabajo que usaremos garantizan esta condición, ya que la suma de los conjuntos de trabajo de todos los procesadores no sobrepasarán en ningún momento los 4GB de memoria con los que configuraremos Ruby.

7.5. Protocolos de coherencia

En cuanto a los protocolos de coherencia de caché a usar, probaremos dos distintos. Son los siguientes:

Protocolo de coherencia MOESI basado en directorio de dos niveles [18]: El uso de directorio para mantener la coherencia proporciona una mayor escalabilidad que otras técnicas como los protocolos de snooping. El nivel de indirección añadido por

el directorio nos permite usar una red de interconexión no ordenada, a diferencia de los protocolos de snooping, donde debe usarse un bus para que todos los procesadores puedan ver las peticiones de memoria. El directorio contiene información sobre el estado de compartición de un bloque en las cachés, información que sirve para determinar las acciones de coherencia a realizar cuando recibimos una petición de memoria a ese bloque. La información típica almacenada en el directorio incluye la lista de compartidores de cada bloque, y un campo que apunta al propietario actual. El directorio puede tomar otras formas, como una lista enlazada de compartidores, o una lista de compartidores con una granularidad mayor a un nodo con un procesador.

En nuestra arquitectura, tenemos dos niveles de directorio. El primer nivel de directorio almacena su información en la caché L2, para así mantener la coherencia entre las cachés L1 de los *tiles* de un chip. Como vimos, la L2 es compartida entre todos los *tiles* de un chip. Cada dirección de bloque de caché se mapea a un *nodo home* que contiene la información de directorio apropiada. El *nodo home* de un bloque será el *tile* a cuya L2 se corresponda su dirección.

El segundo nivel tiene el directorio en memoria DRAM y sirve para mantener la coherencia entre las L2 de los distintos chips de la máquina. En el *tiled* CMP sólo tenemos un chip, por lo que este nivel podría obviarse. En el M-CMP sí cobraría importancia ya que tenemos varios chips y hay que mantener la coherencia entre sus cachés L2.

Si en una solicitud de lectura de un procesador se produce un fallo de caché L1, se envía una solicitud GETS al *nodo home* de la línea de memoria. El controlador de directorio del *nodo home* accederá a la información de directorio, que está en L2, sobre estado de la línea de memoria en cuestión. Si la entrada indica que no hay compartidores, el bloque está libre, y podrá enviarlo a la L1 que solicitó el dato. Además se actualizará la información de directorio para indicar que el nodo que hizo la solicitud es el nuevo propietario. Si el dato no se encuentra en L2 habrá que traerlo de memoria DRAM.

Si, en caso contrario, el directorio indica que hay un nodo que es el propietario del bloque, se enviará a ese nodo un mensaje indicando que envíe directamente los datos al nodo que realizó la solicitud. Además se actualizará la información de directorio añadiendo al nuevo nodo a la lista de compartidores.

Si un GETM (petición de escritura) llega al directorio, y el bloque al que se dirige es compartido por algún nodo, el controlador de directorio envía una invalidación a cada nodo compartidor. Los compartidores responderán con un asentimiento al nodo que hizo la petición. La escritura se completa cuando el nodo que quiere escribir recibe los datos del bloque y los asentimientos de los compartidores.

Protocolo de coherencia MOESI basado en Token [16]: Los protocolos de snooping y basados en directorio necesitan una cuidadosa coordinación de intercambio de mensajes así como las transiciones de una máquina de estados para asegurar la coherencia. Además, las propiedades de la red de interconexión complican aún más el diseño del protocolo para asegurar la coherencia. La coherencia basada en Tokens lo consigue, por el contrario, mediante un sencillo mecanismo de contar e intercambiar tokens.

Se asocia un número fijo de tokens a cada bloque. Para escribir en un bloque, el procesador debe adquirir todos los tokens. Para leer un bloque, sólo es necesario

poseer un token. De esta forma, la coherencia se fuerza contando e intercambiando tokens. Las etiquetas de la caché y los mensajes codifican el número de tokens usando $log_2 T$ bits, siendo T el número de tokens.

La coherencia basada en tokens permite a los procesadores buscar agresivamente los tokens. Se usa una *política de rendimiento* para adquirir los tokens. Por ejemplo, un procesador de un multiprocesador puede predecir qué procesador posee los tokens y enviarle un mensaje directamente sólo a él. Sin embargo, la predicción puede ser incorrecta y la petición puede fallar y no adquirir ningún token. También puede usarse otra política que sea la de lanzar un mensaje de broadcast de manera que el procesador más cercano con tokens pueda enviárselos, reduciendo así la latencia, pero incrementando el uso de ancho de banda, haciendo el protocolo menos escalable. Dependiendo de la implementación del protocolo de tokens pueden usarse la política de rendimiento que más nos interese.

Mientras la política de rendimiento busca maximizar el rendimiento, la coherencia basada en tokens proporciona un *substrato de corrección* para asegurar la coherencia y evitar la inanición, esto es, asegurar que la petición de un procesador se resolverá tarde o temprano. Para esto último, se proporcionan dos tipos de solicitudes: las transitorias y las persistentes. Las primeras son las que pueden fallar, pero las segundas no, evitando al usarlas que haya inanición. El protocolo de token proporciona mecanismos para poder tener peticiones persistentes.

Lo fundamental en un protocolo basado en token es que una simples reglas de conteo de los tokens aseguran que el sistema de memoria se comporte de una manera correcta. Las reglas de conteo son las siguientes:

Conservación de tokens: cada bloque de memoria compartida tiene un número fijo, T, de tokens asociados a él. Una vez se inicialice el sistema, los tokens no pueden crearse ni destruirse. Un token de cada bloque es el token propietario. Este token puede estar limpio o sucio.

Regla de escritura: un componente puede escribir un bloque sólo si tiene todos los tokens de ese bloque y tiene los datos válidos del bloque. Después de escribir en el bloque, el token propietario se establece a sucio.

Regla de lectura: Un componente de un bloque puede leer de un bloque sólo si tiene al menos un token de dicho bloque y los datos válidos.

Regla de transferencia de datos: Si un mensaje de coherencia lleva un token propietario sucio, debe contener los datos del bloque.

Regla del bit de datos válidos: Un componente establece el bit de datos válidos de un bloque cuando llega un mensaje con datos y al menos un token. Un componente limpia el bit de datos válidos cuando ya no tiene en su posesión ningún token. La memoria origen establece el bit de datos válidos siempre que recibe un token propietario limpio, aunque no lleve datos.

Regla de limpieza: Siempre que la memoria recibe el token propietario, establece el estado del token a limpio.

Considerando estas reglas, podemos definir los estados del protocolo de coherencia MOESI dependiendo del número de tokens que un procesador tiene:

Procesadores	16 UltraSPARC-III+ con 4 vías y ejecución en orden. 2 GHz
Caché L1	I&D separados. Tamaño: 128KB. Asociatividad: 4 vías. 64 bytes/línea. Latencia de acceso 2 ciclos.
Caché L2	Tamaño: 1MB cada *slice*. 16MB en total. Asociatividad: 4 vías. 64 bytes/línea. Latencia acceso al array de datos 15 ciclos.
TLB	64 entradas, totalmente asociativo
Memoria RAM	4 GB DRAM. 1 controlador de memoria por chip. 160 ciclos de latencia de memoria + latencia on-chip
Interconexión	Red bidimensional 4x4. Enlaces de 64 bytes. 8 ciclos de latencia por enlace.

Tabla 7.1: Características de la arquitectura objeto

0 tokens:	**I**nvalid (Inválido)
De 1 a $T-1$ tokens, pero no el *token propietario*:	**S**hared (Compartido)
De 1 a $T-1$ tokens, incluyendo el *token propietario*:	**O**wned (Propietario)
T tokens, bit de sucio inactivo:	**E**xclusive (Exclusivo)
T tokens, bit de sucio activo:	**M**odified (Modificado)

Las reglas anteriores aseguran que la coherencia de caché se mantenga, pero no aseguran el avance de la ejecución. La coherencia basada en token evita la inanición lanzando una petición persistente cuando un procesador detecta una inanición potencial. Las peticiones persistentes, al contrario que las peticiones transitorias que se lanzan la mayoría de veces, garantizan que alguna vez habrá éxito. Para asegurar esto, el protocolo de token debe definir como trata con varias peticiones persistentes pendientes.

Un bloque puede encontrarse simultáneamente en, como máximo, T cachés. Doblar T para poder tener el bloque en más cachés sólo implica añadir un bit a los contadores de tokens almacenados en las etiquetas de las cachés y en memoria.

Además, un protocolo de coherencia basado en token evita la necesidad del nivel de indirección adicional que supone el uso de directorio cuando hay una transferencia de datos de una caché a otra. Esto hará que el rendimiento sea superior, siempre que tengamos el suficiente ancho de banda para soportar el protocolo de tokens que utilizaremos, ya que éste usa broadcast para buscar los tokens.

Llamaremos DIRECTORY al protocolo MOESI basado en directorio y TOKEN al protocolo MOESI basado en token.

7.6. Resultados

Las características de la máquina simulada pueden observarse en la tabla de la Tabla 7.1.

Las pruebas se han realizado usando las cargas de la suite SPLASH-2 que pueden verse en la Tabla 7.2. Tenemos un total de diez cargas. Para cada prueba se realizan cinco simulaciones y se realiza la media, para tener en cuenta así la variabilidad en los resultados [2], usando un valor de semilla distinto en cada una de ellas (con lo que conseguimos que las ejecuciones sean diferentes).

Workload	Descripción	Tamaño	Suite	Simulación
Cholesky4x4p	Factorización de una matriz dispersa	tk16.O	SPLASH-2	4 MVs ejecutando Cholesky con 4 procesadores cada una
Radix4x4p	Ordenación de números enteros	1048576 enteros	SPLASH-2	4 MVs ejecutando Radix con 4 procesadores cada una
Ocean4x4p	Simulación del movimiento de los océanos	Océano de 258x258	SPLASH-2	4 MVs ejecutando Ocean con 4 procesadores cada una
Water-NSQ4x4p	Simulación del comportamiento de moléculas de agua a lo largo del tiempo	512 moléculas	SPLASH-2	4 MVs ejecutando Water-NSQ con 4 procesadores cada una
Water-SP4x4p	Como Water-NSQ pero con un algoritmo más eficiente	512 moléculas	SPLASH-2	4 MVs ejecutando Water-SP4 con 4 procesadores cada una
Barnes4x4p	Simulación de la interacción gravitatoria de un sistema de objetos	8192 partículas	SPLASH-2	4 MVs ejecutando Barnes con 4 procesadores cada una
Raytrace4x4p	Obtiene la imagen realista de un modelo mediante trazado de rayos	teapot.env	SPLASH-2	4 MVs ejecutando Raytrace con 4 procesadores cada una.
Tomcatv4x4p	Versión paralelizada del conocido benchmark de la suite SPEC CPU	256	Wind Tunnel	4 MVs ejecutando Tomcatv con 4 procesadores cada una.
Unstructured4x4p	Dinámica de fluidos aplicada a una red desestructurada para modelar una estructura física, como un avión o cuerpo	Mesh.2K	SPLASH-2	4 MVs ejecutando Unstructured con 4 procesadores cada una.
FFT4x4p	Transformada Rápida de Fourier de n números complejos, en una dimensión	16K dobles complejos	SPLASH-2	4 MVs ejecutando FFT con 4 procesadores cada una.

Tabla 7.2: Descripción de las cargas usadas

Una simulación consta de cuatro máquinas virtuales, ejecutando cada una de ellas la misma carga que las demás. Usamos las cuatro arquitecturas explicadas en el Capítulo 7, y para todas ellas también probamos los dos protocolos de coherencia. Esto nos da un total de $10\ cargas \cdot 4\ arquitecturas \cdot 2\ protocolos \cdot 5\ simulaciones = 400\ simulaciones$.

En las Figuras 7.10, 7.11, 7.12, 7.13 y 7.14 puede observarse el resultado de la ejecución de las cargas de trabajo en las cuatro arquitecturas probadas, con los dos protocolos de coherencia utilizados.

Usamos como caso base el *tiled* CMP con mapeo simple, ya que se trata de la arquitectura más básica de las cuatro, y sería la más cercana a una máquina convencional usando virtualización, al no realizarse ningún mapeo especial y usar un *tiled* CMP en que las MVs comparten la caché.

La intención de las otras arquitecturas es mejorar el rendimiento haciendo que los datos accedidos se encuentren más cercanos al procesador que los necesita, reduciendo así la latencia media de las peticiones. Para ello, el mapeo VMCBM trata de llevar los datos accedidos por cada MV a los *slices* L2 de los procesadores usados por la MV, intentando evitar que haya que acceder a otros *slices* de L2 más lejanos. Lo mismo intentamos conseguir usando un M-CMP con mapeo simple, ya que cada MV pasa a tener su propia caché L2 privada. Por su parte, el *tiled* CMP con mapeo TCBM trata de llevar los datos al *slice* L2 del propio procesador que accede al dato siempre que sea posible.

Nos referiremos como **caso base** al *tiled* CMP con mapeo simple, como **Arq2** al M-CMP con mapeo simple, como **Arq3** al *tiled* CMP con mapeo VMCBM y como **Arq4** al *tiled* CMP con mapeo TCBM.

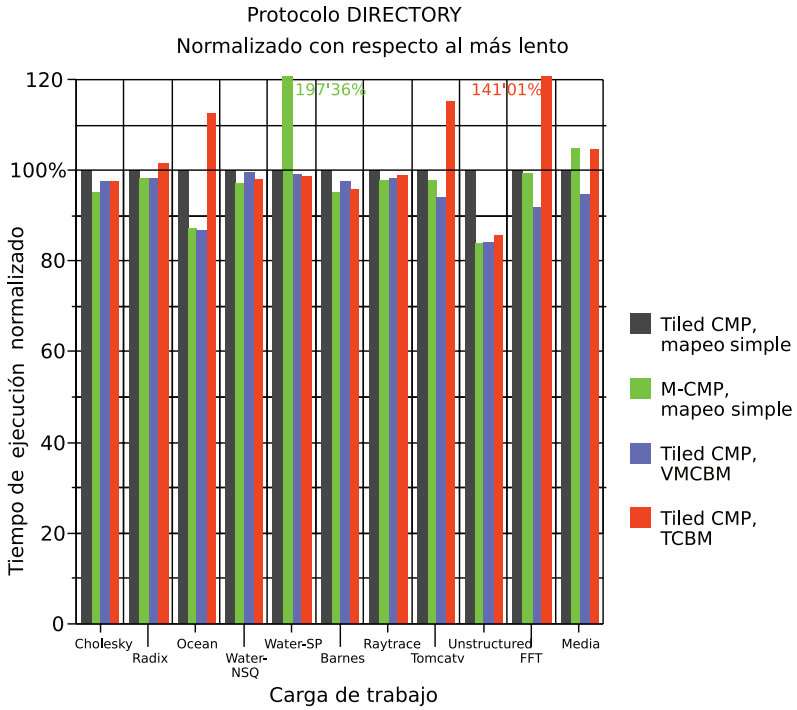

Figura 7.10: Resultados del protocolo DIRECTORY. *Se muestra el rendimiento para cada carga probada, así como la media de rendimiento de cada arquitectura. Los resultados están normalizados, para cada carga, usando como base la arquitectura tiled CMP con mapeo simple. Una barra menor significa más rápido.*

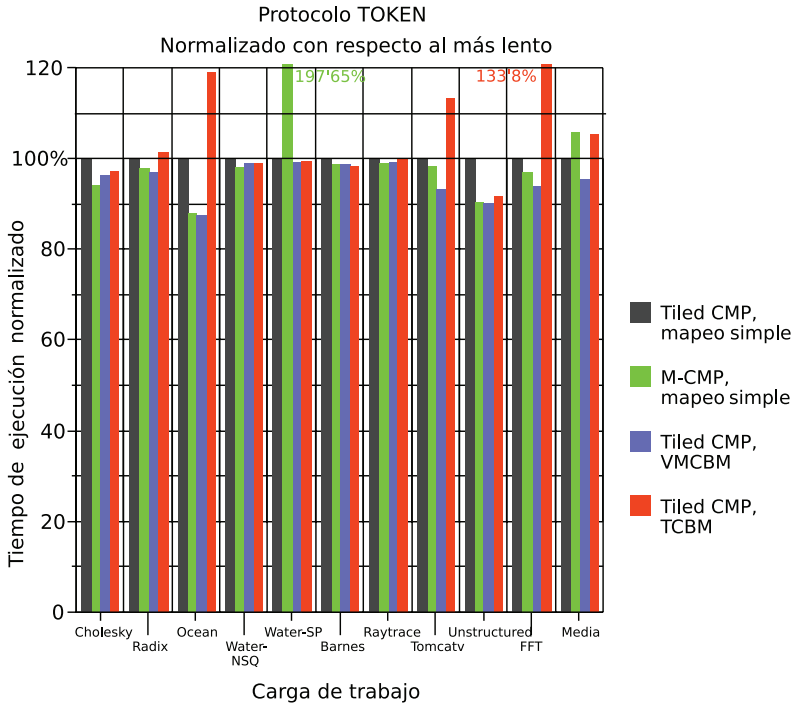

Figura 7.11: Resultados del protocolo TOKEN. *El significado es el mismo que el de la Figura 7.10, pero ahora aplicado al protocolo TOKEN.*

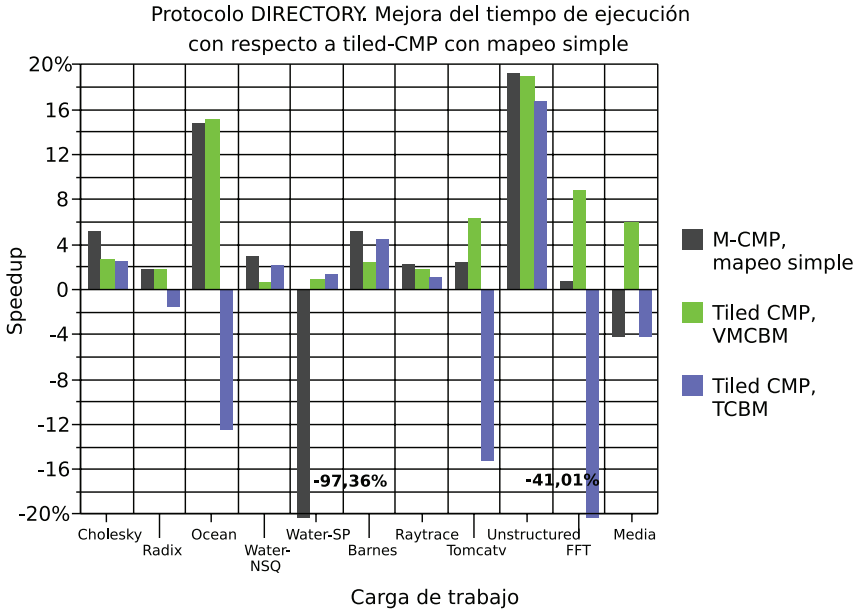

Figura 7.12: Speedup con respecto al caso base con DIRECTORY. *Mostramos, para cada carga de trabajo, la mejora obtenida con respecto al caso base (tiled CMP con mapeo simple), al usar cada una de las otras arquitecturas simuladas. Usamos el protocolo DIRECTORY. Se corresponde con los resultados de la gráfica de la Figura 7.10*

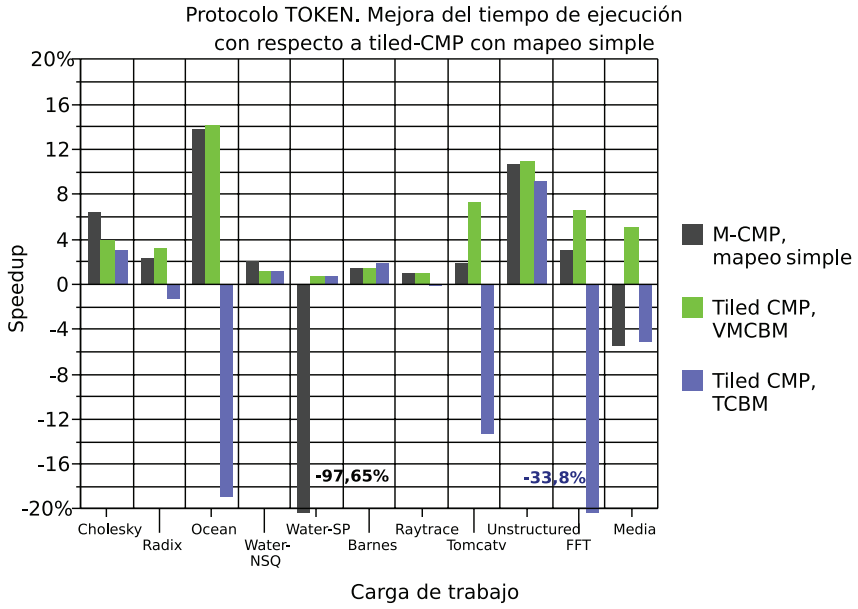

Figura 7.13: Speedup con respecto al caso base con TOKEN. *Mismo significado que la Figura 7.12, pero usando en este caso el protocolo TOKEN. Se corresponde con los resultados de la gráfica de la Figura 7.11*

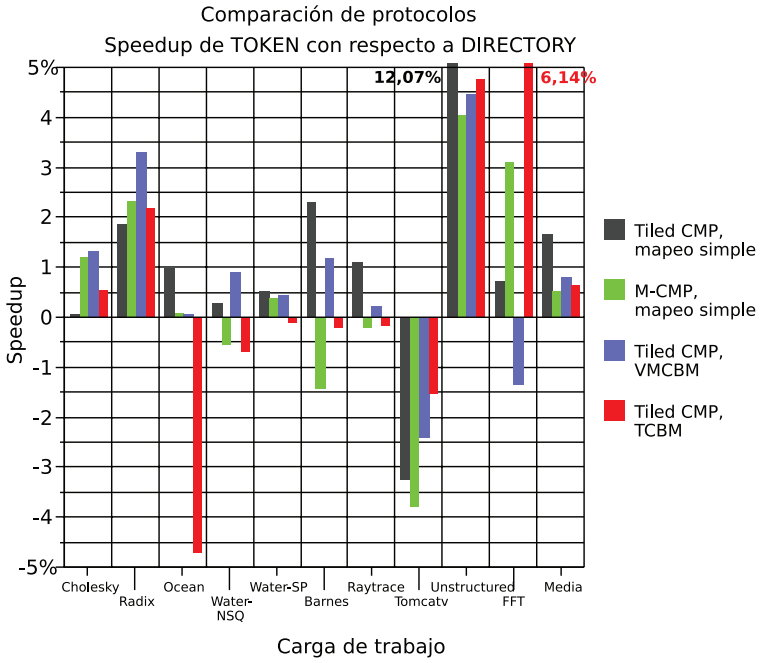

Figura 7.14: Mejora de TOKEN con respecto a DIRECTORY. *Para cada arquitectura y carga de trabajo podemos ver el speedup producido al usar TOKEN en lugar de DIRECTORY como protocolo de coherencia. Finalmente se muestra la mejora media para cada arquitectura.*

De las gráficas podemos obtener varias conclusiones interesantes. La primera y fundamental, es la diferencia de rendimiento existente entre las arquitecturas simuladas. Si nos fijamos en la Figura 7.12, que se corresponde a las pruebas usando el protocolo DIRECTORY, vemos como la Arq3 aporta, de media, una mejora de rendimiento del $5,98\,\%$ con respecto al caso base. Arq2 y Arq4 presentan un rendimiento medio inferior al caso base. En el caso de Arq2, esto se debe a la anomalía que se produce con la carga Water-SP, ya que tenemos un rendimiento un $97,36\,\%$ inferior al del caso base. Si no considerásemos esta anomalía, el rendimiento medio de Arq2 sería un $6,09\,\%$ superior al caso base, es decir, incluso superior al de Arq3.

Arq4 es la arquitectura menos predecible. En Water-SP es la arquitectura que mejor se comporta, sin embargo, su comportamiento errático hace que de media sea un $4,18\,\%$ más lenta que el caso base. Parece que mapear cada bloque al *slice* de caché L2 del procesador que primero accede a él no suele ser buena idea. El problema puede venir dado porque haya un procesador que, antes de nada, acceda a gran parte de los datos para inicializarlos. Esto es algo típico en muchas aplicaciones paralelas. Si es el caso, usando Arq4, todos esos datos se mapearán al *slice* L2 de dicho procesador, desaprovechando los otros *slices* y haciendo que todos los procesadores de la MV deban acceder muy frecuentemente a ese *slice* de L2 cuando se produzca un fallo. Si recodificásemos las aplicaciones para que cada procesador acceda inicialmente a los datos que usará en el procesamiento paralelo (con lo que se mapearían a su *slice*), quizá consiguiésemos mejorar el rendimiento usando Arq4. Los resultados de la carga Water-SP, o la carga Barnes en el caso del protocolo TOKEN (Figura 7.13), en los que Arq4 es la más rápida, hacen pensar que podría ser una buena opción.

Los resultados de las cuatro arquitecturas usando el protocolo TOKEN son similares a los vistos para el protocolo DIRECTORY, no hay más que ver la similitud de las gráficas de mejora de rendimiento de las Figuras 7.12 y 7.13. La mayor mejora se produce en las cargas Ocean (salvo con Arq4, que empeora) y Unstructured. Sí podemos observar que en el caso del protocolo TOKEN, las mejoras de rendimiento al usar las arquitecturas Arq2, Arq3 y Arq4, con respecto al caso base, son algo inferiores a si usamos el protocolo DIRECTORY. Como dijimos, las nuevas arquitecturas pretenden reducir la latencia de los accesos. Vimos como el protocolo de coherencia TOKEN eliminaba un nivel de indirección en los accesos (al eliminar el acceso al directorio), por lo que, al ser ya de por sí una solución más óptima que el protocolo DIRECTORY, la reducción de latencia que podemos obtener es menor, ya que precisamente una de las cuestiones que mejoran las nuevas arquitecturas probadas sería el acceso al directorio, que está en L2, eliminado en el protocolo TOKEN.

Por último, si observamos la Figura 7.14, vemos como, efectivamente, el protocolo TOKEN en promedio es más rápido que el protocolo DIRECTORY. No es una gran mejora, ya que en el caso base, donde es mayor, es de un $1,66\,\%$.

Como vemos, tiene más influencia en el rendimiento el cambio en el mapeo de las páginas que el cambio en el protocolo de coherencia. El cambio de protocolo de coherencia es una operación mucho más compleja, y en teoría, un protocolo más eficiente debe proporcionarnos un mejor rendimiento en las aplicaciones.

Precisamente, Michael Marty et al. [19] muestran como un protocolo basado en Tokens ofrece un rendimiento superior a uno basado en directorio si usamos una arquitectura M-CMP. En nuestro caso concreto de uso de virtualización, vemos como otros cambios más simples, como el del mapeo de la memoria, provocan una mejora superior en el rendimiento.

Por otra parte, los protocolos de coherencia probados son protocolos generales, por así decirlo, en cuanto a que no hacen ningún esfuerzo concreto por adaptarse al uso de virtualización. Si planteamos un nuevo protocolo de coherencia destinado específicamente a máquinas en las que se ejecutará un sistema de virtualización, podríamos optimizarlo a nuestra conveniencia provocando un rendimiento superior al obtenido con los protocolos usados en nuestras pruebas. Este es el caso de los protocolos de coherencia de dos niveles [20] propuestos por Michael Marty y Mark D. Hill. Estos protocolos usan un nivel para mantener la coherencia de caché dentro de cada máquina virtual, y un segundo nivel para la coherencia inter-MV.

8

Conclusiones y vías futuras

Hemos visto la importancia que tiene la simulación en la investigación en arquitectura de computadores. Por otra parte, la virtualización se ha convertido en una técnica casi imprescindible hoy en día en el ámbito de los grandes servidores, permitiendo una mayor versatilidad en el uso del hardware y un mejor aprovechamiento, además de características mejoradas de seguridad, robustez y disponibilidad. Podemos decir que las técnicas de virtualización juegan un papel muy importante en la consolidación de servidores.

Sin embargo, existe un vacío en cuanto a herramientas de simulación que permitan experimentar con las técnicas de virtualización. Una posible solución sería ejecutar software de virtualización dentro del simulador. No obstante, esto presenta problemas como la necesidad de rehacer las cargas de trabajo ya disponibles para su ejecución dentro del software de virtualización (crear las imágenes de las MVs) o la dependencia que se crearía con dicho software.

Introducir la virtualización a nivel hardware dentro del propio simulador es una solución que solventa dichos problemas, además de permitirnos un control mucho mayor a la hora de obtener resultados de rendimiento así como una mejor gestión de los recursos de la máquina que la que nos puede permitir un software de virtualización ya existente (por ejemplo, hemos visto diversas posibilidades de mapeo de la memoria a los *slices* de caché concretos, algo difícilmente obtenible sin contar con soporte hardware).

Los primeros resultados obtenidos son prometedores. Un simple cambio en el mapeo de las páginas reales de las MVs a páginas físicas hace que el rendimiento global de la máquina mejore en un $5,98\,\%$ de media en la ejecución de los diez benchmarks de la suite SPLASH-2 probados. Un cambio de protocolo (operación más compleja que el cambio de mapeo), pasando de un protocolo basado en Directorio a uno basado en Token, también hace que el rendimiento global mejore, en el mejor caso, en un $1,66\,\%$. Si estos cambios en las características de la máquina, que inicialmente tan sólo iban a servir para probar el correcto funcionamiento de Virtual-GEMS, producen tales incrementos de rendimiento, un estudio más en profundidad sobre las necesidades de un sistema virtualizado puede llevarnos a encontrar otras mejoras más influyentes.

Una vez que tenemos las bases de la infraestructura de virtualización, se abren varias vías para su mejora. Lo primero que puede considerarse es ampliar el simulador de manera

que podamos utilizar no sólo los procesadores no segmentados, de ejecución en orden y accesos a memoria bloqueantes de Simics, sino también el procesador superescalar fuera de orden con accesos a memoria no bloqueantes modelado por Opal. Se suele considerar que cuando simulamos la temporización de un sistema de memoria complejo, como es nuestro caso, basta con simular un procesador sencillo para obtener resultados significativos. Sin embargo, Christopher J. Hughes et al. [13] muestran que la diferencia de rendimiento existente entre usar un procesador de este tipo y utilizar un procesador superescalar complejo puede ser impredecible debido a la capacidad de éstos últimos de lanzar múltiples peticiones a memoria concurrentes, algo no posible con los procesadores sencillos simulados por Simics. Por ello sería interesante poder incluir el módulo Opal dentro de la simulación, ya que el incremento de complejidad que deberíamos asumir para llevar la gestión de las peticiones de los procesadores no es muy grande. En lugar de tener una única petición pendiente por procesador deberíamos llevar el control de varias. Por otra parte, los procesadores actuales tienden a ser en orden, por eficiencia energética. Se tiende a explotar más el TLP (paralelismo a nivel de hilo, a través de la replicación de cores sencillos) y menos el ILP (paralelismo a nivel de instrucción, conseguido con mecanismos como la ejecución fuera de orden).

Otra vía interesante es la de conseguir particionado lógico no sólo en el uso de la memoria física, sino también a la hora de repartir los procesadores entre las MVs, así como en los dispositivos de E/S. Hasta ahora, cada procesador se asigna a una única MV, y lo mismo sucede con los dispositivos de E/S, si bien una MV sí puede "contener" varios procesadores o dispositivos de E/S de la máquina. Para conseguir esta granularidad más fina en el reparto de procesadores y E/S necesitaríamos implementar multiplexación en el tiempo de estos recursos, siendo esta una tarea nada trivial.

Otras vías que se abren son las relacionadas con la investigación de nuevas técnicas de virtualización, o más inmediatamente, la investigación en aquellos elementos de la arquitectura cuya modificación pueda mejorar en mayor medida el rendimiento en un sistema virtualizado. Concretamente, el protocolo de coherencia de memoria es un elemento que puede afectar decisivamente en el rendimiento de un sistema de este tipo. Ya existen propuestas de protocolos de coherencia de dos niveles, como las de Michael Marty y Mark D. Hill [20]: un primer nivel para mantener la coherencia dentro de una máquina virtual, y un segundo para las situaciones que no se pueden resolver con dicho primer nivel, es decir, para la coherencia entre las máquinas virtuales.

También sería interesante investigar técnicas que permitiesen compartición de páginas de memoria. El Cellular Disco, por ejemplo, lo permite pero los procesos deben registrar explícitamente las regiones compartidas. Sería interesante poder conseguir que de manera más o menos automática el Hypervisor fuera capaz de detectar páginas que puedan compartirse entre máquinas virtuales (por ejemplo, el código del sistema operativo).

Apéndice A

Uso de Virtual-GEMS

Para realizar simulaciones con nuestra infraestructura de virtualización se han adaptado los scripts que ya se usan en el Grupo de investigación GACOP para ejecutar simulaciones haciendo uso de Simics y GEMS.

Estos scripts están preparados para que podamos elegir la carga de trabajo que deseamos ejecutar en cada simulación; carga a elegir entre todas aquellas que han sido preparadas por el departamento. Los scripts nos permiten lanzar varias simulaciones simultaneamente de manera que estas se situarán en una cola de trabajos y se irán ejecutando en la granja de servidores del departamento conforme vaya habiendo huecos libres para su ejecución.

En nuestro caso una simulación comprenderá la ejecución de la instancia de GEMS, junto con la ejecución de las instancias de SIMICS que hacen las veces de máquinas virtuales.

Para poder usar la infraestructura de virtualización, debemos modificar estos scripts de manera que, para una simulación, ejecuten tanto la instancia de GEMS que hace las veces de servidor, como las instancias de Simics que hacen las veces de clientes. Además debe realizarse la conexión entre clientes y servidor correctamente, de manera que no pueda existir ningún tipo de confusión entre máquinas virtuales y servidores de simulaciones diferentes.

Estos scripts acaban haciendo uso, en las máquinas virtuales, de la interfaz de línea de comandos de Simics. Se usan para ello scripts en Python, que indican a la máquina virtual que inicie el calentamiento, tras lo cual le indican que envíe al servidor la indicación de que queremos limpiar las estadísticas. Una vez hecho esto el script indicará a Simics que debe seguir con la ejecución, y cuando llevemos el número de transacciones de las que consta la ejecución, le indicará a Simics que envíe al servidor la indicación de que queremos obtener las estadísticas, tras lo cual el script le dirá a Simics que envíe la indicación al servidor de que terminó la ejecución de la máquina virtual. Por último, Simics esperará a la respuesta del servidor. De esta forma, una máquina virtual cumple con las fases de la simulación indicadas en el Capítulo 6.10.2.

En el fichero *workloads.py* pueden verse las cargas de trabajo que es posible usar en nuestras simulaciones. El parámetro p en el nombre de la carga suele indicar el número de procesadores que se usarán en la máquina virtual que ejecutará dicha carga. El parámetro n indica el tamaño de la carga a usar. Las principales cargas disponibles son: barnes, cholesky, em3d, fft, jbb, lu, lunc, ocean, radix, radiosity, raytrace, specjbb, tomcatv, unstructured,

volrend, waternsq y watersp. Siguiendo la notación anterior, la carga con nombre fft-p2-n16 ejecutaría un fft con dos procesadores y un tamaño de problema igual a 16. El tamaño y su significado es dependiente de la carga concreta.

El script que nos interesa para configurar las simulaciones a realizar es el que tiene por nombre de fichero *rrencola.sh*. En él tenemos algunas variables que debemos configurar según nuestras necesidades.

En la variable BENCHMARKS indicamos las cargas de trabajo que queremos usar en las simulaciones. Las cargas de trabajo que deseamos usar como máquinas virtuales dentro de una misma simulación, las debemos colocar separadas por coma, pero sin espacios entre ellas. Las cargas de trabajo correspondientes a simulaciones distintas debemos separarlas por espacios. Por ejemplo, consideremos el siguiente valor para la variable BENCHMARKS:

BENCHMARKS="fft-p4-n16-sarek,radix-p4-n1048576-sarek ocean-p4-n258-sarek,barnes-p2-n8k-sarek"

Estaríamos lanzando dos simulaciones, formada cada una de ellas por dos máquinas virtuales. La primera simulación estaría formada por una máquina virtual con 4 procesadores que ejecutaría una fft de tamaño 16, y una segunda máquina virtual con otros 4 procesadores que ejecutaría un radix con tamaño 1048576. La segunda simulación estaría formada por una máquina virtual que ejecutaría un ocean con 4 procesadores y tamaño 258 y otra máquina virtual que ejecutaría un barnes con 2 procesadores y un tamaño de $8k$.

Otras variables a configurar en el fichero *rrencola.sh* son NVM, donde indicaremos el número de máquinas virtuales que deseamos usar en cada simulación, y NPROC_RUBY, donde indicamos el número de procesadores a simular en Ruby en cada simulación. En el caso de la variable BENCHMARK puesto antes como ejemplo, los valores que tomarían estas variables son los siguientes:

NMV="(2 2)"

Indicamos que tenemos dos máquinas virtuales en la primera simulación, y otras dos en la segunda.

NPROC_RUBY="(8 8)"

Indicamos 8 procesadores para cada simulación. La segunda simulación usa en total 6 procesadores (4 una MV y 2 la otra), sin embargo, existe una limitación en Ruby sobre el número de procesadores a usar, que obliga a que ese número sea potencia de dos. Los dos procesadores sobrantes en Ruby no serán usados por ninguna máquina virtual.

Por último, falta por comentar los parámetros que configuran los detalles de la arquitectura simulada en Ruby. Estos se encuentran en la variable RUBY_CONFIG que aparece en el fichero *rrencola.sh*. Esta variable contiene una cadena con el valor de todos estos parámetros. Los más importantes para nuestro caso son:

g_CACHE_DESIGN: Especifica la red de interconexión a usar.

STATIC_CACHE_BANK_DIRECTORY_MAPPING: Cuando vale *true* se activa el mapeo VMCBM. Es un parámetro que aparece nuevo en Virtual-GEMS.

PER_PROC_STATIC_CACHE_BANK_DIRECTORY_MAPPING: Cuando su valor es *true* se activa el mapeo TCBM. También es un parámetro nuevo. Si ni este parámetro ni el anterior están puestos a true, se usará el mapeo simple.

FORCE_SIMPLE_VM_MAPPING: Las posiciones dentro de la red de los procesadores simulados en Ruby sufren un intercambio según estemos simulando un *tiled* CMP (que sólo tiene un chip) o un M-CMP (que tiene varios) debido a la implementación de Ruby. Para realizar correctamente los mapeos indicados en el Capítulo 7.3 para estas dos arquitecturas, *tiled* CMP y M-CMP, es necesario poner esta variable a true si simulamos un M-CMP y a false si simulamos un CMP. Este es un parámetro de configuración nuevo, específico de Virtual-GEMS.

Cuando ejecutemos el script *rrencola.sh*, lo haremos desde el directorio que se generó al compilar Ruby para la arquitectura y protocolo de coherencia concreto que se desea usar. Suponemos que el lector ya conoce el mecanismo usado para realizar esta compilación. Por si no fuera el caso, por ejemplo, si se desea compilar Ruby para simular un CMP usando un protocolo de coherencia basado en directorio debemos colocarnos en la carpeta *$gems/ruby*, siendo *$gems* el directorio en que tenemos instalado Virtual-GEMS, debemos ejecutar el comando *make PROTOCOL=MOESI_CMP_directory DESTINATION=carpetaDestino*. El directorio *$gems/simics/home/carpetaDestino*, que se acaba de crear, es en el que debemos ejecutar el script *rrencola.sh*, que se encuentra a su vez en el directorio *$GEMS/simics-scripts*. La ejecución se hará de la siguiente manera:

rrencola.sh semillaInicial semillaFinal directorioResultados

La semilla es un valor usado para alimentar a la carga de trabajo, de manera que podamos tener ejecuciones distintas para luego realizar la media en los resultados de rendimiento, y así poder mitigar la variabilidad de los resultados en ejecuciones diferentes. Por cada valor entre *semillaInicial* y *semillaFinal* se realizará la ejecución de todas las simulaciones indicadas en *rrencola.sh*.

Los resultados de la simulación se almacenarán en subdirectorios que se crearán en el destino que hayamos indicado con *directorioResultados*. Cada simulación tendrá su propio directorio para los resultados. El nombre del directorio de salida se construye a partir del nombre de las cargas de trabajo usadas en cada simulación. Para la simulación *fft-p4-n16-sarek,radix-p4-n1048576-sarek*, indicada en la variable BENCHMARK del script *rrencola.sh*, se creará el directorio *directorioResultados/fft-p4-n16-sarek,radix-p4-n1048576-sarek_semilla*.

En el directorio de salida de una simulación encontraremos ficheros que contendrán la salida estándar de cada máquina virtual, así como la salida de error (un fichero por máquina virtual y tipo de salida). Ruby además escribirá en el fichero *server.periodic* estadísticas periódicas de la simulación, y en *server.stats* se escribirán las estadísticas finales de toda la ejecución. También tendremos en ese mismo directorio un par de ficheros para la salida estándar y la de error del lanzamiento en sí de la simulación por parte del script *rrencola.sh*, donde podremos comprobar que todo se lanzó correctamente y no hubo ningún problema relacionado con cuestiones como permisos de ejecución o ficheros no encontrados.

La versión de Simics utilizada es la 2.2.19, y la versión de GEMS es la última disponible, la 2.1. El software con Virtual-GEMS ya implementado, así como las instrucciones de instalación, pueden encontrarse en el disco que acompaña a esta memoria.

Bibliografía

[1] Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Mark D. Hill, David A. Wood, and Daniel J. Sorin. Simulating a $2M Commercial Server on a $2K PC. *Computer*, 36(2):50–57, 2003.

[2] Alaa R. Alameldeen and David A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, pages 7–18, 2003.

[3] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, February 2002.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[5] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, volume 28, pages 83–94. ACM Press, May 2000.

[6] Harold W. Cain, Kevin M. Lepak, Brandon A. Schwartz, and Mikko. Precise and accurate processor simulation. *In Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, pages 13–22, 2002.

[7] Sangyeun Cho and Lei Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *MICRO*, pages 455–468, 2006.

[8] J. Emer, P. Ahuja, E. Borch, A. Klauser, Chi-Keung Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, S. Wallace, N. Binkert, A10, R. Espasa, A11, T. Juan, and A12. Asim: a performance model framework. *Computer*, 35(2):68–76, 2002.

[9] Ayose Falcón, Paolo Faraboschi, and Daniel Ortega. Combining Simulation and Virtualization through Dynamic Sampling. In *IEEE International Symposium on Performance Analysis of Systems and Software 2007*, pages 72–83. IEEE Computer Society, April 2007.

[10] Ricardo Fernández and José M. García. RSIM x86: A Cost-Effective Performance Simulator. In *19th European Conference on Modelling and Simulation*, pages 774–779, Riga, Latvia, June 2005. European Council for Modelling and Simulation.

[11] S. Gill. The Diagnosis of Mistakes in Programmes on the EDSAC. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 206(1087):538–554, May 1951.

[12] Kingshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262, 2000.

[13] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: simulating shared-memory multiprocessors with ILP processors. *Computer*, 35(2):40–49, 2002.

[14] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[15] Mario D. Marino. 32-core CMP with multi-sliced L2: 2 and 4 cores sharing a L2 slice. pages 141–150, 2006.

[16] Milo M. K. Martin. *Token coherence*. PhD in Computer science, University of Wisconsin - Madison, 2003.

[17] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005.

[18] Michael R. Marty. *Cache coherence techniques for multicore processors*. PhD in Computer science, University of Wisconsin - Madison, 2008.

[19] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M. K. Martin, and David A. Wood. Improving Multiple-CMP Systems Using Token Coherence. *hpca*, 00:328–339, 2005.

[20] Michael R. Marty and Mark D. Hill. Virtual hierarchies to support server consolidation. *SIGARCH Comput. Archit. News*, 35(2):46–56, May 2007.

[21] B. Nayfeh and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[22] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[23] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer System Simulation: The SimOS Approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[24] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[25] Manuel E. Acacio Sánchez. *Improving the Performance and Scalability of Directory-based Shared-Memory Multiprocessors*. PhD in Computer science, Universidad de Murcia, February 2004.

[26] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Microarchitecture, 2002. (MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*, pages 271–282, 2002.

[27] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. The Liberty Simulation Environment, Version 1.0. *Performance Evaluation Review: Special Issue on Tools for Architecture Research*, 31(4), March 2004.

[28] Virtutech. *Simics Programming Guide 2.2.19*. August 2005.

[29] Virtutech. *Simics Reference Manual 2.2.19*. August 2005.

[30] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[31] J. J. Yi and D. J. Lilja. Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations. *Transactions on Computers*, 55(3):268–280, 2006.