Análisis de modelos de memoria en plataformas de virtualización. Formalización de un prototipo funcional de plataforma con Cache y TLB.

Jesús Mauricio Martín Chimento

<checholcc@gmail.com>

Legajo: C-4058/4

Universidad Nacional de Rosario.

Facultad de Ciencias Exactas, Ingeniería y Agrimensura.

Tesina de Grado - Licenciatura en Ciencias de la Computación

Director : Carlos Luna

<cluna@fing.edu.uy>

Co-directores : Gustavo Betarte

<gustun@fing.edu.uy>

Juan Diego Campo

<jdcampo@fing.edu.uy>

Resumen

La virtualización es una técnica que se utiliza para correr múltiples sistemas operativos en una sola máquina física, pero creando la ilusión de que en realidad cada uno de estos sistemas operativos corre dentro de una Máquina Virtual diferente. El Monitor de Máquinas Virtuales es el encargado de administrar los recursos compartidos por los sistemas operativos que corren en las Maquinas Virtuales de manera que todos se puedan ejecutar adecuadamente. En particular, el acceso y uso de la Memoria Principal es un aspecto crítico que el Monitor de Máquinas Virtuales debe controlar.

En este trabajo se presenta un modelo formal idealizado de un Monitor de Máquinas Virtuales, sobre el cual se demuestran propiedades que garantizan el uso correcto de las acciones provistas por dicho Monitor a los distintos sistemas operativos virtualizados y el correcto acceso y uso de la Memoria Principal por parte de estos sistemas operativos. Se utiliza el asistente de pruebas de Coq para las demostraciones y posterior extracción de una versión ejecutable del modelo.

Agradecimientos

En el cierre de una etapa muy importante en mi vida, quisiera recordar y agradecederles a muchas de las personas que me dan y que me han dado fuerzas para que hoy haya llegado a donde estoy.

Primero que nada, quiero agradecerle a mi mamá, que siempre me apoyó, me bancó con todo y que contribuyó en gran medida para hoy sea la persona que soy; a mi hermana, que tuvo y tiene que soportar mis locuras; bueno, a mi familia en general, quienes siempre estuvieron y me alentaron en las buenas y en las malas.

Le doy gracias a mis compañeros lccianos, pero por sobre todas las cosas mis amigos, con los que compartí un aula, recreos y horas de estudio, entre muchas otras cosas. En especial a Fede y el Negro (a.k.a. Emanuel), con quienes arranqué luchándola desde el primer día; Eze, Taihu, Peggy, Tincho, Pablo B., Chacho, Fer, Coco, Fabri, Ema y Zeta, a quienes conocí al poco tiempo de comenzar la carrera y rápidamente entablamos una buena amistad; Bibi y Uciel, mis compañeros viajeros; Juanito, German, Exe, Gustavo, Guido G. y Andrea, a quienes conocí nioniando; y a Cristian y Luis, quienes pese a no haber sido compañeros míos me dieron una gran mano en mis aventuras por el viejo continente.

Les doy gracias a mis docentes, quienes con sus enseñanzas y su buena onda me guiaron en mis estudios y comenzaron a formarme como profesional. En especial, le doy gracias a Dante y Guido, dado que gracias a ellos comencé a descubrir y adentrarme en el espíritu de la LCC; y a Fidel, quien me ayudó a dar los primeros pasos por el camino de losas amarillas de la docencia y me enseño a mover mis alas para que hoy estén listas para volar.

Fuera del ámbito académico, le doy gracias a mis grandes amigos Fede y el Pollo (a.k.a. Gonzalo), quienes, pese a que en los últimos tiempos no podamos vernos de manera muy seguida, siempre estarán disponibles cuando los necesite (y viceversa); y a Naza y el Cartu, grandes

compañeros de aventuras, con quienes viví muchas situaciones anecdóticas.

Por último, agradecerles a Carlos, Gustun y Juan Diego dado que, de no haber sido por su ayuda y su guía, este trabajo no habría podido realizarse en su plenitud.

Checho

Índice general

Capítulo 1

Introducción

1.1. Virtualización

El concepto de ***Virtualización*** surge a principio de los '60 de la mano de IBM. Dicho concepto consiste en abstraer los recursos de un sistema informático, ocultando las características físicas del sistema a usuarios y aplicaciones. De esta manera, IBM buscaba obtener un uso más eficiente de los grandes y costosos mainframes que utilizaban en esa época.

Desde un punto de vista más práctico, la virtualización es una técnica que se utiliza para correr múltiples sistemas operativos, llamados ***guest OSs*** (sistemas operativos invitados), en una sola máquina física, pero creando la ilusión de que en realidad cada uno de los guest OS corre dentro de una ***Máquina Virtual*** (o Virtual Machine, o **VM**) diferente. Dichas **VMs** son una abstracción de la *máquina física* (i.e. la computadora). Estas son provistas a cada guest OS por una capa delgada de software llamada ***Monitor de Máquina Virtual*** (o Virtual Machine Monitor, o **VMM**). Además, el **VMM** es el encargado de administrar los recursos compartidos por las **VMs** (e.g. Memoria Principal).

Estructura de la Virtualización

La estructura de la virtualización suele ser presentada en forma de estratos, como se la ilustra en la figura 1.1. En dicha figura *Apps.* son las aplicaciones que utilizan los guest OSs, *Hardware* es la abstracción que ven los guest OSs del hardware físico (*Hardware real*) y *Host OS* es el sistema operativo corriendo en la máquina física.

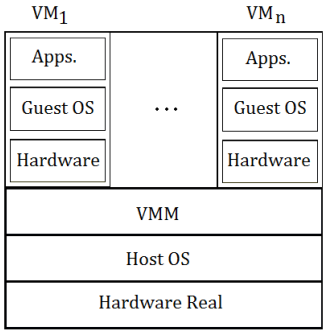

Figura 1.1: Estructura de la Virtualización

Estilos de Virtualización

Históricamente hay 2 estilos de virtualización: ***Fullvirtualization*** y ***Paravirtualization***. En el primer estilo, cada **VM** es una replica exacta del hardware de la computadora, lo que hace posible que el software y el sistema operativo corran en la **VM** de la misma manera en que lo harían en el hardware de la computadora. En el segundo estilo, cada **VM** es una versión simplificada del hardware de la computadora. Los sistemas operativos que se instalan en las **VMs** (i.e. los guest OSs) son adaptados para que corran adecuadamente. Además, los guest OSs pueden ser divididos en dos clases:

- guest OSs confiables
- guest OSs no confiables

Los guest OSs confiables pueden acceder directamente a las instrucciones privilegiadas (protegidas) y corren en modo supervisor. En cambio, los guest OSs no confiables estos corren en modo usuario. Cuando estos últimos quieran realizar una operación privilegiada (protegida), en su lugar realizarán una ***hypercall***[1] la cual será atendida por el **VMM**. Cabe destacar que las aplicaciones de los guest OSs corren en un nivel de privilegios inferior al de los guest OSs no confiables.

[1]Una hypercall es para un guest OS lo que una llamada a sistema es para un sistema operativo normal.

1.2. Un modelo idealizado de plataforma de virtualización

Un modelo es un conjunto de conceptos los cuales definen a un sistema. Estos incluyen la descripción de los elementos que componen al sistema, pueden describir el comportamiento de dichos elementos y definen las relaciones y propiedades que presentan, entre otras cosas. Ahora bien, cuando hablamos de un modelo idealizado estamos haciendo referencia a un modelo sobre el cual se hicieron algunas suposiciones sobre sus elementos para que su definición sea más fácil de realizar.

Dado lo anterior, podemos decir que un modelo idealizado de plataforma de virtualización es un conjunto de conceptos que la definen, sobre el cual se han hecho ciertas suposiciones para facilitar su definición.

Por último, vale mencionar que cuando el modelo se describe utilizando un lenguaje formal (e.g. matemática, lógica) dicha descripción es llamada *especificación del sistema*. Además, describir modelos utilizando lenguajes formales permite el estudio de propiedades del sistema con un nivel de rigurosidad mayor al que tendríamos si lo hiciéramos con un lenguaje corriente.

1.3. Diseño de un VMM por máquina de estados

Este trabajo se enmarca dentro del proyecto de investigación VirtualCert[2]. Dicho proyecto tiene como objetivo principal desarrollar un modelo idealizado de plataforma de virtualización focalizándose en la especificación y verificación formal de determinadas propiedades de seguridad las cuales son deseables que sean garantizadas por las distintas plataformas de virtualización. En particular, persigue modelar formalmente la interacción de diferentes sistemas operativos que se estén ejecutando sobre una misma plataforma virtualizada y establecer cuáles son los mecanismos que garantizan determinadas propiedades de no interferencia, particularmente en relación a los datos manejados por los sistemas que se ejecutan concurrentemente sobre esa plataforma.

En [10] se presenta parte de la especificación antes mencionada. La misma está descripta en COQ [28] y fue diseñada en base a una máquina de estados. Esto último significa que el **VMM**

[2]proyecto VirtualCert: Hacia una plataforma de virtualización certificada, de la Agencia Nacional de Investigación e Innovación (ANII, Uruguay). `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

está asociado en cada instante a un cierto estado. Dicho estado irá modificándose a partir de la ejecución de las distintas acciones que hayan sido especificadas en el modelo. Por supuesto, luego de la ejecución de una acción cualquiera el sistema debe estar asociado a un estado válido, por lo que en la especificación se introdujo el concepto de estado válido, el concepto de ejecución válida y , en base a ellos, se demostró formalmente que las posibles ejecuciones válidas mantienen invariante la validez del estado actual del sistema. Esto último es de gran importancia para el modelo dado que permite probar que este cumple con otras propiedades de interés, como por ejemplo isolation (o aislación) de guest OSs a nivel de ejecución de una acción. En otras palabras, la ejecución de una acción en un guest OS no depende ni se ve afectada por el estado de los demás guest OSs.

Cabe destacar que el modelo de este proyecto está basado en Xen ARM (v5) [3]. Por lo tanto, sigue la línea de la paravirtualización. Además, el **VMM** diferencia guest OSs confiables de no confiables.

1.4. Objetivos del trabajo

El objetivo principal de este trabajo es analizar modelos de memoria en plataformas de virtualización y formalizar un prototipo funcional de un modelo idealizado de plataforma que contemple, en particular, el uso de Cache y TLB[3]. Puntualmente:

1. Estudiar el estado del arte del uso de modelos de memoria en plataformas de virtualización, incluyendo el uso de Cache y TLB.

2. A partir del trabajo [10] y de una extensión propuesta con el uso de Cache y TLB, obtener un prototipo funcional certificado, usando Coq, de un modelo de plataforma que incluya en particular el uso de Cache y TLB.

1.5. Contribuciones

En primer lugar, a partir del estudio del estado del arte se confeccionó un *reporte técnico*[4] [11] que habla acerca de los distintos componentes de memoria (e.g. RAM, Cache, TLB) que se

[3]Translation Lookaside Buffer o, en español, Buffer de Traducción Adelantada

[4]En el capítulo 2 de este trabajo se presentará un resumen de dicho reporte.

encuentran en una computadora, de cómo influye en ellos el uso de memoria virtual y de cómo interactúan las plataformas de virtualización con los mismos.

Segundo, se extendió el modelo idealizado de plataforma de virtualización actual del proyecto VirtualCert agregando el uso de Cache y TLB al estado. Para esto, se modeló la estructura de la Cache y la TLB, fueron implementadas operaciones para agregar y quitar elementos de ellas y fue probado que las mismas son correctas según su especificación, se analizó el hecho de que dichas operaciones no pueden ser independientes del estado y fueron reimplementadas las distintas acciones de la plataforma (sin y con manejo de errores) para que estas tengan en cuenta que se está haciendo uso de Cache y TLB en el nuevo estado. Este último hecho llevó a tener que demostrar que dichas implementaciones cumplen con su semántica (i.e. son correctas). Cabe destacar que las reimplementaciones antes mencionadas mantienen la propiedad de invarianza del estado válido luego de su ejecución. Sin embargo, la demostración de lo anterior, la cual puede ser consultada en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php` (en el modulo COQ **Exec_invariant.v**), fue hecha fuera de este trabajo.

Finalmente, se pudo extraer del nuevo modelo una versión certificada del **VMM**, mediante la inferencia de código ejecutable brindada por COQ.

1.6. Trabajos Relacionados

A continuación se listan algunos proyectos relacionados directa o indirectamente con este trabajo.

Verisoft XT project

Verisoft XT[5] es un proyecto financiado por el Ministerio Federal Alemán de Educación e Investigación (BMBF) y administrado por el Centro Aeroespacial Alemán (DLR).

Su objetivo principal es el de verificar formalmente la correcta funcionalidad de sistemas de computación. Dicha corrección de los sistemas es probada matemáticamente con la ayuda de computadores. Este último hecho busca evitar que haya error humano en las pruebas.

Dentro de este proyecto pueden encontrarse trabajos como [2], el cual verifica formalmente la corrección de un **VMM** pequeño; o como [18], el cual busca verificar la corrección del **VMM** de la plataforma de virtualización Microsoft Hyper-V^{TM} utilizando VCC [7].

[5] `http://www.verisoftxt.de`

PROSPER project

Prosper[6] es un proyecto financiado por la Fundación Sueca de Investigación Estratégica (SSF) y desarrollado de manera colaborativa entre el Instituto Sueco de Ciencias de la Computación (SICS) y el grupo de Ciencias de la Computación Teórica del Instituto Real de Tecnología (KTH[7] - Estocolmo, Suecia).

Su objetivo es modelar y verificar técnicas y herramientas seguras para **VMM's**. Dichos *VMM's* serán utilizados en sistemas embebidos.

Como dicho proyecto ha comenzado recientemente todavía no hay publicaciones que involucren al mismo.

seL4 project

seL4[8] es un proyecto desarrollado por el Centro Nacional Australiano de Investigación en Tecnologías de Información y Comunicación (NICTA).

Su objetivo principal es la verificación del microkernel L4, demostrando propiedades de isolation sobre una implementación de un sistema operativo, llamado seL4, para dicho microkernel.

Dentro de este proyecto pueden encontrarse numerosas publicaciones, siendo [15] la más destacada dado que obtuvo el premio a mejor publicación en "Proceedings of the 22nd ACM Symposium on Operating Systems Principles". Dicho trabajo, según afirman sus autores, fue el que mostró la primera prueba de corrección funcional formal de un sistema operativo de kernel completo.

1.7. Estructura del informe

En el capítulo 2 se presenta un resumen de [11], reporte técnico acerca del estado del arte del uso de modelos de memoria en plataformas de virtualización, incluyendo el uso de Cache y TLB; en el capítulo 3 se presenta el modelo idealizado del **VMM** definido en el proyecto VirtualCert; en el capítulo 4 se agregará al modelo la descripción de la Cache y de la TLB, y se hará mención de los cambios que se producen al realizar dicha modificación; en el capítulo 5 se mostrará la implementación de las operaciones de la Cache y la TLB, la implementación de las acciones

[6] http://www.sics.se/projects/prosper
[7] http://www.kth.se/?l=en_UK
[8] http://ertos.nicta.com.au/research/sel4/esunproyecto

de la plataforma (sin y con manejo de errores), sus pruebas de corrección y se derivará código ejecutable certificado a partir de las pruebas antes mencionadas; en el capítulo 6 se analizará de que manera influiría en el modelo la introducción de ciertos cambios en su especificación; y en el capítulo 7 se darán las conclusiones de este trabajo y se mencionarán posibles trabajos a futuro.

Capítulo 2

Memoria en Entornos de Virtualización

2.1. Memoria Principal

2.1.1. Conceptos Básicos

A la hora de utilizar virtualización, uno de los recursos a abstraer de una computadora es la Memoria Principal (o memoria **RAM** o, como será mencionada de aquí en más, simplemente *memoria*). Dicha abstracción puede no ser una tarea sencilla de realizar.

La memoria es un arreglo de 2^n **palabras** (o bytes) las cuales poseen su propia dirección (única). Además, como puede verse en la figura 1, esta consta de **bloques** de K palabras, siendo 1 el menor tamaño de los anteriores. [25]

A la hora de trabajar con multiprogramación y/o con multitarea, el sistema operativo debe ofrecer alguna forma de administración de la memoria de tal manera que al tener más de un proceso en ejecución (utilizando el espacio de usuario), estos operen de manera correcta.

Las maneras más comunes de administrar la memoria son las siguientes:

- Segmentación de Memoria (o Segmentation)
- Paginación de Memoria (o Paging)

Por otra parte los accesos a la memoria pedidos por el micro son manejados por la Unidad de Gestión de Memoria (o **MMU** por sus siglas en inglés).

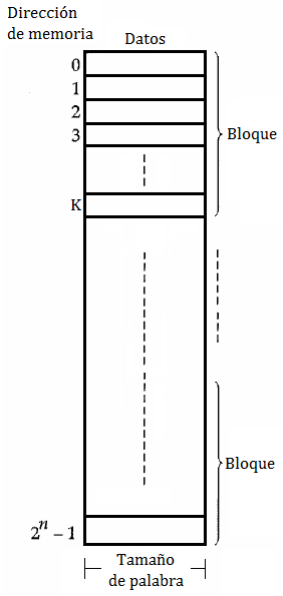

Figura 2.1: Memoria Principal

2.2. Memoria Cache

2.2.1. Conceptos Básicos

Un problema que puede surgir al trabajar con la memoria es que como el incremento de velocidad de los micros no se ha dado de manera equitativa al incremento de la velocidad de la antes mencionada, estos son muchos más rápidos. Este hecho hace que las memorias no sean lo suficientemente rápidas para almacenar y transmitir los datos que los micros necesitan, generando retardos considerables, dado que los micros tienen que esperar que las memorias estén disponibles para poder trabajar.

Una posible solución al problema antes mencionado radica en no utilizar un único componente de memoria sino en utilizar una memoria lenta de gran capacidad y una memoria de poca capacidad pero con tiempos de acceso rápidos, haciéndolo de la siguiente manera:

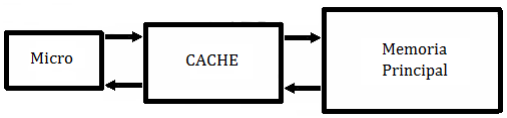

Figura 2.2: Ubicación de la CACHE

la memoria de gran capacidad será la memoria y la otra memoria, que recibirá el nombre de Memoria Cache, o simplemente Cache, estará ubicada entre la memoria y el micro, siendo la Cache la clave para equiparar la velocidad de la memoria con la velocidad del micro.

La Cache hará las veces de cache de memoria para la memoria, de ahí su nombre. Cada una sus entradas está compuesta por un ***índice*** que identifica a que sección de la Cache se está haciendo referencia (hay S secciones, donde $S << 2^n$), un ***bloque*** de la memoria y una ***etiqueta*** que identifica la dirección de dicho bloque [25]. Cuando el procesador intente acceder a un dato de la memoria, se comprobará primero si el mismo está en la Cache. De ser así, se dice que se ha producido un ***Cache hit*** y el dato será enviado al micro. Caso contrario, se dice que se ha producido un ***Cache miss*** y se agrega a la Cache el bloque de la memoria que contenga al dato en cuestión. Obviamente, como la Cache tiene poca capacidad, sus datos pueden tener que ser desalojados con el fin de agregar nuevos datos a la misma.

2.2.2. Diseño de la Cache

A la hora de diseñar una Cache hay que tener en cuenta los siguientes aspectos [25]:

- Tamaño de la Cache.

  El ***tamaño*** de la Cache viene limitado por el coste de este tipo de memorias, por lo que estas suelen ser de poca capacidad. Esto hace que los tiempos de acceso a la misma sean rápidos. Además, a menor capacidad de Cache, mayor velocidad de acceso.

- Función de Correspondencia.

  Cuando se agrega a la Cache un nuevo bloque de datos, la ***función de correspondencia*** es la encargada de determinar la posición en la cual este se ubicará. Los tipos de funciones de correspondencia usuales son:

  Correspondencia Directa: Al bloque i-ésimo de la memoria le corresponde siempre el índice ***i módulo S*** en la Cache, donde S es la cantidad de secciones de la Cache e i es la dirección del bloque.

  Correspondencia Asociativa: Cualquier bloque de la memoria puede ser colocado en cualquiera de las S secciones de la Cache.

Correspondencia Asociativa por Conjuntos: La Cache se divide en k conjuntos de bloques. Luego, al bloque i-ésimo de memoria le corresponde el conjunto ***i módulo k***. Dicho bloque podrá ubicarse en cualquier posición de ese conjunto.

- Algoritmo de Reemplazo.

  Al agregar un bloque a la Cache puede que otro bloque que ya resida en esta tenga que ser reemplazado debido a que o bien un bloque reside en la posición correspondiente al nuevo bloque, o bien la Cache está llena. Dicha tarea se lleva acabo por el ***Algoritmo de Reemplazo***. Los algoritmos de reemplazo más comunes son:

  FIFO (First In First Out): Este algortimo consiste en asociar a cada bloque de datos la hora en la que han sido alojados en la Cache. Luego, cuando un bloque de datos en la antes mencionada deba ser reemplazado, se elegirá al más viejo. [23]

  En otras palabras, y como su nombre lo indica, el primer bloque de datos en entrar a la Cache será el primer bloque de datos en salir de la misma.

  RAND (Random): Este algoritmo consiste simplemente en elegir el bloque de datos a reemplazar en la Cache de manera aleatoria.

  LFU (Least Frequently Used): Este algoritmo consiste en asociar un contador diferente a cada bloque de datos. Dichos contadores se incrementarán cada vez que se haga referencia a su bloque de datos asociado. Luego, cuando un bloque de datos en la Cache deba ser reemplazado, se elegirá el bloque al que se hallan hecho menos referencias (i.e. que tenga el contador más bajo). [23]

  LRU (Last Recently Used): Este algoritmo consiste en asociar a cada bloque de datos la hora en la cual han sido utilizados por última vez. Luego, a la hora de realizarse el reemplazo de un bloque de datos en la Cache, se elige el bloque que hace más tiempo que no ha sido utilizado. [23]

- Política de Escritura.

Si el contenido de un bloque que se encuentra en la Cache es modificado, entonces dicho bloque debe ser escrito nuevamente en la memoria (particularmente antes de ser reemplazado). La ***política de escritura*** es quien marca en que momento debe realizarse la operación de escritura en la memoria. Existen (por lo menos) 2 técnicas para esto:

Escritura Inmediata (o Write-through): Cuando una operación de escritura se lleva acabo entonces la información escrita en la Cache se transmite inmediatamente a la memoria.

Además, la memoria siempre contendrá una copia *up-to-date* (actualizada) de toda la información del sistema y una copia válida del total de sus estados en cualquier momento dado. Esto permite que si el micro falla, el sistema pueda restaurarse de manera más sencilla. [24]

Escritura Demorada (o Copy-back o Write-Back): Cuando una operación de escritura se lleva acabo entonces solo se modifica la Cache. Dicho cambio será transmitido a la memoria más tarde, cuando ocurra un *Cache miss*.

Esta técnica puede complicar la lógica de la Cache. Ahora se necesitará un bit especial, llamado ***dirty bit***, para saber cuando se debe copiar información desde la Cache a la memoria. [24]

2.3. Cache y Memoria Virtual

2.3.1. Conceptos Básicos

La Memoria Virtual (o Virtual Memory) es una técnica que se utiliza para darle a los programas la ilusión de que la memoria es más grande de lo que en realidad es. La misma consiste en que los procesos utilicen un conjunto de direcciones diferentes a las del ***espacio de direcciones físicas***. Dicho conjunto recibe el nombre de ***espacio de direcciones virtuales***. [8]

Como cada una de las direcciones virtuales que estén siendo utilizadas estará asociada a una dirección física, hay que proveerles a los programas un mecanismo para ***traducir*** una dirección virtual a una dirección física cuando se acceda a la información en la memoria. Dicho mecanismo

consiste en revisar una tabla alojada en memoria la cual es creada al mismo tiempo en que se crea el proceso que utilizará las direcciones virtuales. Esta tabla recibe el nombre de ***Tabla de Paginación***. En la misma se almacenan las distintas traducciones dirección virtual-dirección física. Además, el **MMU** es el encargado de realizar dichas traducciones. Cabe destacar que, como a cada proceso se le asigna un espacio de direcciones virtuales único, existirá una Tabla de Paginación por cada proceso que este corriendo [20].

Un problema que puede surgir al usar Memoria Virtual es que, como el espacio de direcciones virtuales puede ser mucho mayor que el espacio de direcciones físicas, habrá casos en los que todas las partes de un proceso a cargar no quepan en memoria. Dado este problema, se utiliza un espacio de almacenamiento secundario llamado ***área de intercambio*** en el cual se almacenan las partes que no sean alojadas en la memoria [25]. Cuando sea necesario utilizar una parte de un programa que este en el disco, simplemente será intercambiada por una de las partes alojadas en memoria.

2.3.2. Fallo de página

Cuando un programa haga referencia a una dirección virtual cuya dirección física asociada no esté en la memoria, el **MMU** hará que el micro levante una trap. Dicha trap recibe el nombre de ***fallo de página*** (o ***page fault***). Luego, el sistema operativo deberá realizar una de las siguientes tareas:

- Si existe un bloque libre en la memoria, traer desde el disco a dicho bloque la página a la que se está haciendo referencia y después actualizar la Tabla de Paginación.
- Si la memoria está llena, escoger alguna de las páginas alojadas en memoria, copiar su información a disco (solo si esta no ha sido actualizada antes de haberse producido el fallo de página), liberar la dirección física asociada a la página escogida, traer a memoria desde el disco la información relacionada a la nueva página a alojar en memoria y actualizar la Tabla de Paginación [23].

2.3.3. Tipos de Cache

Según como sean los índices y las etiquetas de la Cache (físicos y/o virtuales) esta es llamada [20]:

- Físicamente Indexada, Físicamente Etiquetada (o **PIPT** por sus siglas en inglés)
- Virtualmente Indexada, Virtualmente Etiquetada (o **VIVT** por sus siglas en inglés)
- Virtualmente Indexada, Físicamente Etiquetada (o **VIPT** por sus siglas en inglés)

En el primer caso, las direcciones que la Cache ve ya han sido traducidas por el **MMU** (i.e. son direcciones físicas). Dado lo anterior, las Caches PIPT también son llamadas Caches Físicas.

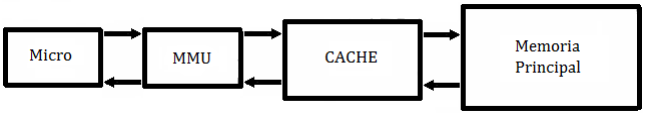

Figura 2.3: Cache Física

En el segundo caso, las direcciones que la Cache ve aún no han sido traducidas por el **MMU** (i.e. son direcciones virtuales). Dado lo anterior, las Caches VIVT también son llamadas Caches Virtuales. En este caso, la traducción de una dirección virtual solo será necesaria en caso de producirse un Cache miss.

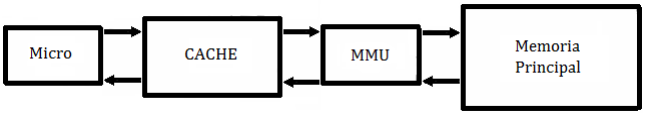

Figura 2.4: Cache Virtual

Desafortunadamente, pese a que eliminar la necesidad de traducir una dirección en un Cache hit aumenta el rendimiento de la computadora, el uso de una Cache VIVT puede complicar el diseño y/o trabajo de sistemas operativos multitareas y multiprocedadores [9]. Esto se debe a la posible aparición tanto de ***sinónimos de direcciones*** como de ***homónimos de direcciones*** en la Cache. Los primeros consisten en la asignación de nombres de dirección de distintos espacios de direcciones virtuales a la misma página. Los segundos consisten en la asignación del mismo nombre de dirección para direcciones pertenecientes a espacios de direcciones virtuales diferentes.

En el tercer y último caso, se busca conseguir las ventajas de performance que ofrecen las Caches indexadas virtualmente junto con las ventajas que ofrece la simpleza de la arquitectura de las Caches etiquetadas físicamente. Por ejemplo, este tipo de Caches no tienen el problema de la posible aparición de homónimos de direcciones en ellas dado que cada dirección física tiene

un TAG único en la Cache [20]. Además, la búsqueda de un bloque en la Cache y la traducción de la dirección virtual pueden realizarse de manera simultánea.

2.4. TLB

2.4.1. Conceptos Básicos

Como la traducción de una dirección virtual a una dirección de la memoria suele ser un proceso un poco costoso el cual se realiza cada vez que un proceso necesite acceder a memoria y que es muy probable que un programa acceda a la misma dirección varias veces, el mecanismo de traducción de direcciones viene acompañado de un buffer, el cual recibe el nombre de **Buffer de Traducción Adelantada**, o simplemente **TLB** por sus siglas en inglés (*Translation Lookaside Buffer*).

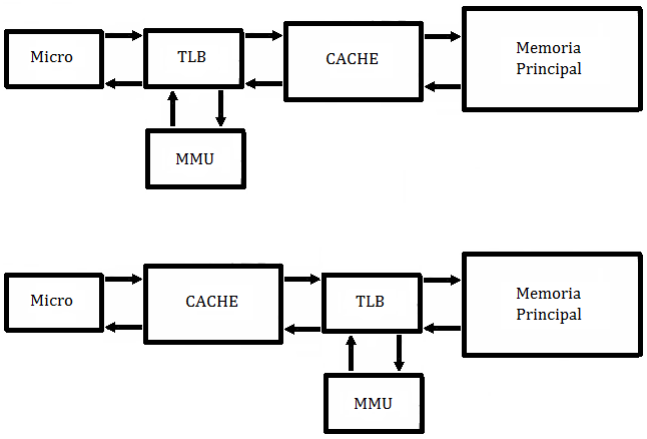

Figura 2.5: TLB Cache

Ahora, antes de pedirle la traducción de una dirección virtual al **MMU**, primero se chequeará si la misma se halla en la TLB sea cual sea el tipo de Cache que se utilice. Si la traducción de la dirección virtual se halla en la TLB, lo que generalmente se llama ***TLB hit***, no solo no se deberá realizar dicha traducción sino que también el acceso a la misma será rápido ya que el **MMU** tomará la información directamente de la TLB. Por otro lado, cuando una traducción no se halle en la TLB, lo que generalmente se llama ***TLB miss***, el **MMU** o el sistema operativo serán los encargados de realizar dicha traducción dependiendo del tipo de TLB que se este utilizando (manejada por hardware o manejada por software) [27]. Una vez realizada

la nueva traducción esta será almacenada en la TLB. En caso de no haber lugar en la TLB para almacenar una nueva traducción, se emplean los mismos Algoritmos de Reemplazo que se utilizan en la Cache para proceder con dicho almacenamiento.

Cabe destacar que a lo igual que las Caches VIVT, la TLB siempre está expuesta a la posibilidad de que en ella se hallen tanto sinónimos de direcciones como homónimos de direcciones. Para evitar este problema, cada vez que se produce un cambio de contexto la TLB se vacía. Además, la TLB siempre utiliza una función de correspondencia Asociativa [25].

2.5. Memoria, Cache y TLB en entornos de Virtualización

2.5.1. Virtualización de la Memoria

A la hora de crear una **VM**, uno de los recursos compartidos que debe ser virtualizado (abstraído) por el **VMM** es la memoria. Esta no suele ser una tarea sencilla, especialmente cuando se utiliza Memoria Virtual.

Un guest OS ejecutado dentro de una **VM** espera un espacio de direcciones físicas *zero-based*[1] [30]. Para darle a cada **VM** la ilusión de que su espacio de direcciones físicas es zero-based, la memoria usualmente es virtualizada agregando un nivel extra de direcciones (y de traducción de direcciones).

Utilizando la terminología empleada en [5], llamaremos ***dirección de máquina*** a una dirección de la memoria, ***dirección física*** a una dirección del nuevo nivel de direcciones agregado a las **VMs** y ***memoria física***[2] a dicho nivel (o espacio) de direcciones.

Además, si el guest OS emplea memoria virtual, aparte del traductor de direcciones "físicas" a direcciones de máquina (y viceversa), habrá que agregar un traductor de direcciones virtuales a direcciones "físicas" (y viceversa).

Por último, cabe destacar que a la hora de virtualizar la memoria, hay que tener en cuenta lo siguiente [22]:

i) La región de memoria del **VMM** debe estar protegida (aislada) de las regiones de memoria de las de los kernels de los guest OSs y/o las aplicaciones de usuario.

[1] El elemento inicial de la memoria se encuentra en el índice cero

[2] Para evitar confusiones con la memoria real, cuando se haga referencia a la memoria física o a una dirección de dicha memoria, escribiremos memoria/dirección "física".

II) La región de memoria de los kernels de los guest OSs debe estar protegida de las regiones de memoria de las aplicaciones de usuario.

III) La región de memoria de una aplicación de usuario debe estar protegida de las otras regiones de memoria de las demás aplicaciones de usuario.

2.5.2. Virtualización de la Cache

A la hora de trabajar con virtualización, la Cache no debe abstraerse. En cambio, los guest OSs utilizan la Cache como si fuera la Cache propia de su máquina sin saber que todos ellos están utilizando en realidad la misma Cache. EL **VMM** es el encargado de que dicho uso de la Cache sea transparente para los distintos guest OSs y de que se realice de forma correcta.

2.5.3. Virtualización de la TLB

De la misma manera que con la Cache, los guest OSs utilizarán la TLB como si fuera la TLB de su propia máquina, por lo que este aspecto no debe ser abstraído. Las que si deben ser abstraídas son las Tablas de Paginación y su manejo.

Algunas plataformas optan por utilizar ***estructuras shadow***[3] en el **VMM** y dejar que este se encargue de la comunicación con el hardware a la hora de la traducción. Otras permiten a los guest OSs comunicarse directamente con el hardware, aunque imponen ciertas restricciones en el uso de las páginas.

2.5.4. Problemas al virtualizar

Pese a que al utilizar virtualización se obtienen muchos beneficios, esta también puede enfrentarnos a nuevos problemas. Dos ejemplos de lo antes mencionado son el diseño de la arquitectura x86 y Thrashing.

Diseño de la arquitectura x86

Los procesadores de la familia Intel x86 no fueron diseñados para ser virtualizados. Por ejemplo, estos cuentan con un conjunto de 17 instrucciones extremadamente sensibles para ser co-

[3]Las estructuras shadow son copias de estructuras que contienen datos privilegiados, los cuales no pueden ser manejados directamente por un guest OS. Estas son mantenidas por el **VMM** para simular ciertas instrucciones.

rridas en un ambiente de virtualización las cuales no emiten traps [21]. Esto hace que dichas instrucciones no puedan ser virtualizadas dado que el **VMM** no puede detectarlas.

Las plataformas de estilo *paravirtualization* lidian con dichos problemas de diseño restringiendo o eliminando el uso de las partes no virtualizables de la arquitectura x86 en los guest OSs (e.g. Denali no ofrece a las **VMs** memoria virtual).

Las plataformas de estilo *fullvirtualization* tienen una tarea más complicada a la hora de lidiar con lo anterior. Dado que no pueden realizar ninguna modificación sobre los guest OS estas deben implementar una serie de algoritmos un tanto complejos para lograr virtualizar la arquitectura x86.

Thrashing

En una máquina con varias **VMs** corriendo en ella, las cuales a su vez se encuentran corriendo un buen número de aplicaciones puede ocurrir que si la máquina host posee poca memoria o si esta posee una buena cantidad de memoria pero las aplicaciones de las **VMs** requieren el uso de mucha memoria, se produzca Thrashing (o Hiperpaginación).

Se llama Thrashing a la situación en la cual la cantidad de recursos utilizados al trabajar crece pero la cantidad de trabajo a realizar disminuye. Normalmente este término se utiliza para hacer referencia a cuando se alojan y desalojan, en forma sucesiva y constante, páginas de un proceso en la memoria y/o el área de intercambio. Esto causaría que la performance del sistema operativo host o de las **VMs** caiga estrepitosamente.

Capítulo 3

Modelo Idealizado de Plataforma de Virtualización

En este capítulo, en primer lugar se dará una reseña de la notación utilizada para describir la formalización del modelo en general. Luego, dada la gran relevancia de la memoria con respecto a este trabajo, se describirá el modelo de la misma. Finalmente, se procederá a describir la formalización del modelo en general.

3.1. Notación Utilizada

Expresiones lógicas

Para las expresiones lógicas son utilizados los operadores estándar ($\vee, \wedge, \Rightarrow, \neg, \exists, \forall$). Además, este tipo de expresiones serán escritas en cursiva.

Ejemplo: $\forall(n1\ n2\ n3 : \mathbb{N}), n1 = n2 \Rightarrow n2 = n3 \Rightarrow n1 = n3$

Tipos

Cuando se haga referencia a una definición de tipo, a una variable, a un campo de un tipo estructurado, o al nombre de un tipo, también se utiliza letra cursiva para nombrarlos. Además, llamaremos *Prop* al tipo que representa al conjunto de todas las proposiciones.

Mappings

- En este trabajo los mappings son considerados como funciones parciales. En particular, el dominio debe tener definida una relación de igualdad entre sus elementos para poder evaluar la función.

- El tipo de un mapping se define igual que una función estándar, es decir, dando el nombre seguido de dos puntos y los tipos de dominio y co-dominio.

  Ejemplo: *map-padd-to-madd: padd* $\rightarrow$ *madd*

- Los mappings se implementaron utilizando listas. Sin embargo, cuando hagamos referencia a un mapping vacío utilizaremos el valor *map-empty* en lugar de *nil* (lista vacía). De todas formas, dichos valores son iguales.

- Para acceder al valor de un mapping (i.e. evaluar la función) se utilizan corchetes.

  Ejemplo: *p2m[pa1]* evalúa el mapping *p2m* para el valor *pa1*.

  El resultado puede ser:

  - *Error pa1*, si el mapping no tiene asignado ningún valor para *pa1*.
  - *Value ma1*, si el mapping sí tiene asignado un valor para *pa1*, el cual es *ma1*.

- El mapping *g = (p2m[pa1] := ma)* es un mapping idéntico a *p2m* a excepción del valor asociado a la entrada pa1, cuyo valor es *ma*.

- Un elemento de un mapping tiene la siguiente forma: *Item indice valor-asociado*

  Para acceder a sus componentes se utilizan las siguientes funciones:

  indice (Item ind val) = ind

  pagina (Item ind val) = val

Records

- Los records se definen usando llaves, dentro de las cuales cada uno de sus campos se separan por una coma. Un campo es indicado por una etiqueta (i.e. nombre) y el tipo dicho campo.

  Ejemplo: *Record-date* $:= \{day : \mathbb{N}, month : \mathbb{N}, year : \mathbb{N}\}$

- El acceso a un campo de un record se realiza mediante un punto seguido por el nombre del campo.

  Ejemplo: *r.campo1*

  En el ejemplo se accede al campo *campo1* del record *r*.

- Para indicar que dos records son iguales con excepción de algunos de sus campos se utiliza la relación $\widehat{=}_{id_1,...,id_n}$, donde $id_1, ..., id_n$ son los nombres de las etiquetas cuyos valores varían.

  Ejemplo:

  fecha1-1985 := {*day* := *4*, *month* := *7*, *year* := *1985*}

  fecha2-1985 := {*day* := *8*, *month* := *10*, *year* := *1985*}

  fecha1-1985 $\widehat{=}_{day,month}$ *fecha2-1985*

  Ahora bien, si al comparar los valores anteriores quisieramos dejar explícito en la relación cuales son los valores que varían en *fecha2-1985* respecto de *fecha1-1985* podemos hacerlo de la siguiente manera.

  fecha1-1985 $\widehat{=}_{day:=8,month:=10}$ *fecha2-1985*

Tipos inductivos

Los tipos inductivos se definen dando su nombre y sus constructores, estos últimos de a uno por guarda, como se muestra en el siguiente ejemplo:

Ejemplo: *Tree a* :=

$$|\,\textit{Leaf: } t \rightarrow \textit{Tree } a$$
$$|\,\textit{Node: } t \rightarrow \textit{Tree } a \rightarrow \textit{Tree } a \rightarrow \textit{Tree } a$$

Pattern Matching

A la hora de analizar el valor de una variable inductiva, lo haremos haciendo pattern matching sobre la misma.

Ejemplo: ***match*** *itree* ***with***

$$|\,\textit{Leaf } n \Rightarrow e_1$$
$$|\,\textit{Node } n \textit{ lt } rt \Rightarrow e_2$$

end

En el ejemplo anterior, si la variable inductiva *itree:Tree a* es una *Leaf*, entonces se procede a analizar la expresión e_1. En cambio, si la variable antes mencionada es un *Node*, entonces se procede a analizar la expresión e_2.

Funciones

A la hora de definir una función, lo haremos de la siguiente manera:

foo (a:A) (b:B) : C := e a b

donde *foo* es el nombre de la función a definir, a y b son los argumentos que toma dicha función, C es el tipo del valor que devuelve la función antes mencionada y e es una expresión de tipo $A \to B \to C$.

3.2. Introducción al Modelo de Memoria

Uno de los componentes a modelar del estado, el cual es de gran importancia a la hora de trabajar con virtualización, es la memoria. El **VMM** no solo debe encargarse de que la gestión de la memoria por parte de los guest OSs sea realizada de manera correcta sino que también este debe proveerles mecanismos para asegurar que los anteriores utilicen la memoria de manera confidencial, i.e. que un guest OS no utilice una dirección de memoria asignada a otro a guest OS.

3.2.1. Direccionamiento de la Memoria

El modelo de la memoria se describe tal cual se muestra en la figura 3.1. El mismo involucra tres tipos de direcciones y dos tipos de mappings: direcciones de máquina, direcciones "físicas", direcciones virtuales, mappings de direcciones virtuales a direcciones "físicas" y mappings que asocian direcciones "físicas" con direcciones de máquina.

Direcciones de Máquina

Como ya se explicó en la sección 2.5, este tipo de direcciones corresponden a las direcciones reales (i.e. físicas) de la memoria, provistas por el hardware.

Por lo general, al trabajar en virtualización suele hacerse referencia a la memoria llamándola *memoria de máquina*. Por lo tanto, este tipo de direcciones serán representadas con el tipo abstracto *madd*.

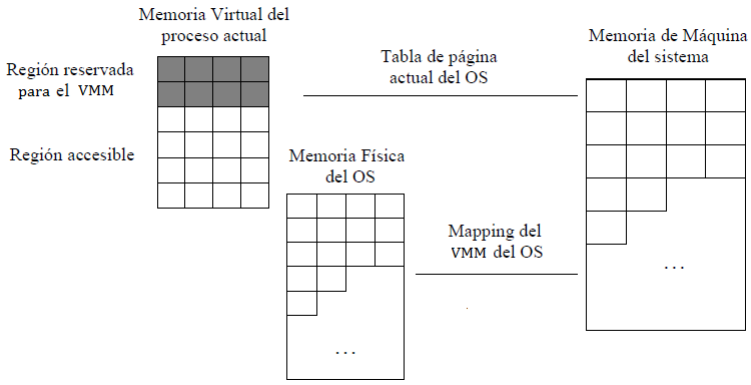

Figura 3.1: Esquema de direccionamiento de la memoria

Direcciones "Físicas"

Como ya se explicó en la sección 2.5, las direcciones "físicas" corresponden a la abstracción de las direcciones de máquina que utilizan los guest OSs.

Al espacio de direcciones "físicas" de un guest OSs lo llamaremos *memoria "física"*. Por lo tanto, este tipo de direcciones serán representadas con el tipo abstracto *padd*.

Direcciones Virtuales

Estas direcciones, provistas por el *MMU*, son las utilizadas por las aplicaciones que corren en los guest OSs. El espacio de direcciones virtuales de cada aplicación será llamado *Memoria Virtual*. Por lo tanto, este tipo de direcciones serán representadas con el tipo abstracto *vadd*.

Cabe destacar que cada uno de los guest OSs tiene una porción de su espacio de direcciones virtuales reservada para uso del **VMM**.

Mappings de direcciones virtuales a direcciones de máquina

Estos mappings corresponden a las tablas de paginación. Los mismos tienen la siguiente forma:

$$page\text{-}table := vadd \rightarrow madd$$

Como ni las aplicaciones ni los guest OSs no confiables tienen permiso para leer o escribir en las tablas de paginación, dichas tareas deben realizarse en modo supervisor. Además, el modelo debe garantizar que toda dirección de máquina mapeada en una tabla de paginación de un determinado guest OSs esté asignada a dicho OS.

Cabe destacar que, como cada sistema operativo cuenta con primitivas para asociar direcciones virtuales con direcciones "físicas", el *VMM* designa de forma interna la dirección de máquina que corresponda. Esto se verá con más detalle más adelante cuando se presenten las acciones *new* y *delete*.

Mappings de direcciones "físicas" a direcciones de máquina

En este modelo, este tipo de mappings son mantenidos por el **VMM**. Los mismos reflejan la memoria de máquina que tiene asignada un guest OS, y asocian la memoria "física" que maneja dicho OS con dicha memoria de máquina. Estos mappings tienen la siguiente forma:

$$padd\text{-}to\text{-}madd := padd \rightarrow madd$$

Los mismos pueden ser modificados utilizando las acciones *page pin* y *page unpin*. Dichas acciones serán abordadas con más detalle más adelante cuando se presenten todas las acciones.

Cabe destacar que en en la descripción del modelo estos mappings son llamados *p2m* por sus siglas en inglés (physical to machine).

3.2.2. Contenido de la Memoria

La memoria es representada con un mapping, con el siguiente tipo:

$$system\text{-}memory := madd \rightarrow page$$

Una *page* (i.e. página) consiste en un record cuyos campos hacen referencia al contenido de la página y a quién es el dueño de dicha página. El contenido de la página pueden ser valores de lectura/escritura, una tabla de paginación o nada (en el caso de que la página no haya sido inicializada). El dueño de una página puede ser un guest OS, el *VMM* o puede no tener dueño. Formalmente,

option v :=

$|$ *None: option v*

$|$ *Some: v* $\rightarrow$ *option v*

content :=

$|$ *RW: option Value* $\rightarrow$ *content*

$|$ *PT: page-table* $\rightarrow$ *content*

$|$ *Other: content*

page-owner :=

$|$ *Hyp: page-owner*

$|$ *OS: os-ident* $\rightarrow$ *page-owner*

$|$ *No-Owner: page-owner*

page := {*page-content: content, page-owned-by: page-owner*}

3.3. Formalización del modelo

El estado de la plataforma consiste en un record cuyas componentes serán descriptas en detalle a continuación. Previo a dichas descripción serán introducidos algunos conceptos involucrados en la misma.

3.3.1. Sistemas Operativos

Cada guest OS es representado por un identificador de tipo *os-ident*. Dicho identificador es utilizado para acceder a distintas características que el guest OS al que representa posee. Por ejemplo, el estado posee información acerca de la dirección "física" de la tabla de paginación actual que esté utilizando un proceso de un guest OS y si se encuentra alguna hypercall, realizada por el guest OS, pendiente (i.e. información de ejecución). Esto es,

$$os := \{curr\text{-}page{:}padd,\ hcall{:}option\ hypercall\}$$

Dicha información es accedida mediante el uso de un mapping con el siguiente tipo:

$$oss\text{-}map := os\text{-}ident \rightarrow os$$

Otro ejemplo de su uso es la designación de direcciones de máquina para las direcciones físicas de cada guest OS. La misma viene dada por un mapping con el siguiente tipo:

$$hypervisor\text{-}map := os\text{-}ident \rightarrow padd\text{-}to\text{-}madd$$

3.3.2. Modos de Ejecución

Este modelo distingue entre 2 modos de ejecución: modo usuario y modo supervisor. Los mismos son utilizados como mecanismos de protección dado que las instrucciones privilegiadas solo pueden ser ejecutadas en modo supervisor y en este modelo solo los guest OS confiables y el *VMM* corren en dicho modo. Si un guest OS no confiable necesita realizar una operación que sea privilegiada, este tiene que pedirle al *VMM* que realice dicha operación en su nombre.

Los modos de ejecución son formalizados mediante el siguiente tipo:

$$\begin{aligned} exec\text{-}mode := \\ &| usr\text{:}\ exec\text{-}mode \\ &| srv\text{:}\ exec\text{-}mode \end{aligned}$$

Cabe destacar que, en un momento dado, solo puede haber un guest OS activo (o *aos*) o esperando que el *VMM* atienda una hypercall invocada por este, de entre todos los guest OS que estén corriendo. Esto se refleja con el siguiente tipo:

$$\begin{aligned} os\text{-}activity := \\ &| running\text{:}\ os\text{-}activity \\ &| waiting\text{:}\ os\text{-}activity \end{aligned}$$

3.3.3. Estado y Contexto

Estado

El estado guarda la información necesaria para describir la situación actual del **VMM** y los guest OSs que corren en el mismo. El mismo se describe de la siguiente manera:

$$\begin{aligned} state := \{&active\text{-}os\text{:}\ os\text{-}ident, \\ &aos\text{-}exec\text{-}mode\text{:}\ exec\text{-}mode, \\ &aos\text{-}activity\text{:}\ os\text{-}activity, \\ &oss\text{:}\ oss\text{-}map, \\ &hypervisor\text{:}\ hypervisor\text{-}map, \\ &memory\text{:}\ system\text{-}memory\} \end{aligned}$$

donde,

- **active-os:** Indica cual es el guest OS activo.
- **aos-exec-mode:** Representa el modo de ejecución actual del procesador.
- **aos-activity:** Representa la actividad del guest OS actual.
- **oss:** Contiene la información de ejecución de los guest OSs.
- **hypervisor:** Contiene la designación de memoria de máquina de los guest OSs.
- **memory:** Representa la memoria del sistema.

Contexto

El contexto del modelo guarda la clasificación de los guest OSs en confiables y no confiables; y la clasificación de las direcciones virtuales entre accesibles y no accesibles (por los guest OSs). Dicha información es invariante en el sistema. Formalmente,

$$context := \{ctxt\text{-}vadd\text{-}accessible:\ vadd \rightarrow bool,\ ctxt\text{-}oss:\ os\text{-}ident \rightarrow bool\}$$

donde,

- **ctxt-vadd-accessible:** Indica qué direcciones virtuales son accesibles por los guest OSs.
- **ctxt-oss:** Indica cuáles guest OSs son confiables y cuáles no.

Cabe destacar que, en el modelo, el contexto es accedido desde cualquier expresión, como si se tratara de una variable global. Además, una dirección no accesible por los guest OSs es accesible por el **VMM**.

3.3.4. Estado Válido

Como ya ha sido mencionado en la sección 1.3, el **VMM** está asociado en cada instante a un cierto estado. Ahora bien, dichos estados deben cumplir con ciertas propiedades para ser considerados estados válidos. Estas propiedades involucran a las componentes del estado y al contexto, y son resumidas en el predicado *valid-state*. Dicho predicado consiste en las siguientes propiedades:

- **trusted-os-not-hypercall:** Los guest OSs confiables no deben realizar llamadas al **VMM**.
- **running-os-not-hypercall:** El guest OS actual no se puede ejecutar si tiene pendiente una llamada al **VMM**.
- **valid-hyper-exec-mode:** El **VMM** siempre se debe ejecutar en modo supervisor.
- **valid-trusted-os-exec-mode:** Los guest OSs confiables también deben correr en modo supervisor.
- **valid-untrusted-os-exec-mode:** Los guest OSs no confiables corren en modo usuario.
- **valid-hypervisor:** Las direcciones de máquina que son asignadas a un guest OS por medio del mapping del **VMM**, deben pertenecerle a dicho guest OS.
- **valid-virtual-mapping:** Si una dirección virtual es accesible por un guest OS, debe pertenecer al espacio de direcciones virtuales de dicho OS. Si no, dicha dirección debe ser accesible por el **VMM**.
- **valid-current-page:** La página asociada al campo current-page de cada guest OS debe pertenecer a él, y ser de tipo PT (i.e. tabla de paginación).
- **injective-hyper-mappings:** El **VMM** no puede asignarle a dos direcciones "físicas" diferentes la misma dirección de máquina, para cualquier guest OS.
- **va-has-valid-pa:** La dirección de máquina asignada a una dirección virtual en una tabla de paginación debe estar asignada a alguna dirección "física" por medio del mapping de **VMM**.

3.3.5. Acciones

Como ya se mencionó anteriormente, el sistema modela una máquina de estados que transiciona entre un estado y el siguiente por medio de la ejecución de una acción. Estas acciones son representadas con el siguiente tipo:

action :=

$|$ *silent: action*

$|$ *read: vadd* $\rightarrow$ *action*

| *read-hyper: vadd → action*

| *write: vadd → value → action*

| *write-hyper: vadd → value → action*

| *new-untrusted: os-ident → vadd → padd → action*

| *new-trusted: vadd → padd → action*

| *new-hyper: vadd → madd → action*

| *del-untrusted: os-ident → vadd → action*

| *del-trusted: vadd → action*

| *del-hyper: vadd → action*

| *switch: os-ident → action*

| *lswitch-trusted: padd → action*

| *lswitch-untrusted: os-ident → padd → action*

| *hcall: Hyperv-call → action*

| *ret-ctrl: action*

| *chmod: action*

| *page-pin-trusted: padd → ptype → action*

| *page-unpin-trusted: padd → action*

| *page-pin-untrusted: os-ident → padd → ptype → action*

| *page-unpin-untrusted: os-ident → padd → action*

Las mismas, se describen a continuación:

silent

No realiza cambios observables en el estado del sistema.

read (va:vadd)

Un guest OS lee la dirección virtual (accesible) *va*.

read-hyper (va:vadd)

El **VMM** lee la dirección virtual (no accesible) *va*.

write (va:vadd, val:value)

Un guest OS escribe en la dirección virtual (accesible) *va* el valor *val*.

write-hyper (va:vadd, val:value)

El **VMM** escribe en la dirección virtual (no accesible) *va* el valor *val*.

new-untrusted (osi:os-ident, va:vadd, pa:padd)

Un guest OS no confiable con identificador *osi* solicita, hypercall mediante, un nuevo espacio de memoria para su proceso actual. Para esto, agrega una entrada a su tabla de paginación actual con valores *va* y *ma*, donde *ma* es la dirección de máquina asociada a la dirección "física" *pa* de *osi* en el mapping del **VMM** y *va* una dirección virtual accesible.

new-trusted (va:vadd, pa:padd)

Un guest OS confiable solicita un nuevo espacio de memoria para su proceso actual. Para esto, agrega una entrada a su tabla de paginación actual con valores *va* y *ma*, donde *ma* es la dirección de máquina asociada a la dirección "física" *pa* del guest OS activo en el mapping del **VMM** y *va* una dirección virtual accesible.

new-hyper (va:vadd, ma:madd)

El **VMM** solicita un nuevo espacio de memoria no accesible para el proceso actual del guest OS activo. Para esto, agrega una entrada a la tabla de paginación actual con valores *va* y *ma*, donde *va* es una dirección virtual no accesible y *ma* una dirección de máquina.

del-untrusted (osi:os-ident, va:vadd)

Un guest OS no confiable con identificador *osi* libera un espacio de memoria en su proceso actual, a través de una hypercall quitando de su tabla de paginación actual la entrada con valor *va*, donde *va* es una dirección virtual accesible.

del-trusted (va:vadd)

Un guest OS confiable libera un espacio de memoria en su proceso actual quitando de su tabla de paginación actual la entrada con valor *va*, donde *va* es una dirección virtual accesible.

del-hyper (va:vadd)

El **VMM** libera un espacio de memoria en su proceso actual quitando de su tabla de paginación actual la entrada con valor *va*, donde *va* es una dirección virtual no accesible.

switch (osi:os-ident)

El VMM cambia el guest OS activo por el guest OS con identificador *osi*.

lswitch-untrusted (osi:os-ident, pa:padd)

Un guest OS no confiable con identificador *osi*, hypercall mediante, cambia de proceso actual. Para esto, cambia la dirección de su tabla de paginación actual por la dirección "física" *pa*.

lswitch-trusted (pa:padd)

Un guest OS confiable cambia de proceso actual modificando la dirección de su tabla de página actual por la dirección "física" *pa*.

hcall (hyp:Hyperv-call)

Un guest OS no confiable pide al **VMM** que ejecute el servicio privilegiado *hyp*.

ret-ctrl

Un guest OS le cede el control al **VMM**.

chmod

El **VMM** le devuelve el control al guest OS no confiable luego de atender su hypercall pendiente.

page-pin-trusted (pa:padd, pt:ptype)

Un guest OS confiable registra, para su propio uso, la página de memoria que corresponde a la dirección "física" *pa*. Esta es clasificada como de tipo *pt*.

page-pin-untrusted (osi:os-ident, pa:padd, pt:ptype)

Un guest OS no confiable registra, hypercall mediante y para su propio uso, la página de memoria que corresponde a la dirección "física" *pa*. Esta es clasificada como de tipo *pt*.

page-unpin-trusted (pa:padd)

La página de memoria correspondiente a la dirección "física" *pa* es des-registrada por un guest OS confiable.

page-unpin-trusted (osi:os-ident, pa:padd)

La página de memoria correspondiente a la dirección "física" *pa* es des-registrada por un guest OS no confiable con identificador *osi*.

3.3.6. Semántica de las Acciones

El comportamiento de las acciones se específica con una precondición *Pre* y una poscondición *Post*. La precondición indica qué restricciones se deben cumplir en el estado sobre el cual se quiere ejecutar una acción, para poder realizar dicha ejecución. La poscondición indica qué características se deben cumplir en el estado obtenido luego de haber ejecutado una acción.

Formalmente, podemos describir las anteriores de la siguiente manera:

$$Pre := state \rightarrow action \rightarrow Prop$$
$$Post := state \rightarrow action \rightarrow state \rightarrow Prop$$

,donde *Pre* toma el estado sobre el cual se quiere ejecutar una acción, la acción en cuestión y define las restricciones que se deben cumplir en forma de predicado; y *Post* toma el estado sobre el cual se ejecutó una acción, la acción en cuestión, el estado obtenido luego de la ejecución de la acción y define las características que se deben cumplir en el estado obtenido luego de haber ejecutado la acción en forma de predicado.

A continuación se dan algunos ejemplos. En estos ejemplos *s'* representa al estado obtenido luego de ejecutarse la acción en cuestión. El resto de las pre y poscondiciones puede consultarse en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`, en el modulo COQ **Exec.v**.

Acción Read

Pre (s, read (va)) :=

context.os-accessible[va] = True $\wedge$ *s.aos-activity = running* $\wedge$

$\exists$ *(ma:madd), va-mapped-to-ma (s, va, ma)* $\wedge$

page-type (s.memory[ma]) = RW

Post (s, read (va), s') :=

s = s'

La precondición de esta acción pide que la dirección virtual que se quiera leer sea accesible por los guest OSs, que tenga asociada una dirección de máquina que sea de tipo lectura/escritura (*RW*) y que el guest OS activo se esté ejecutando.

La poscondición de esta acción pide que, como su ejecución no cambia el estado, el estado sobre el cual se ejecuta la misma sea igual al estado obtenido luego de dicha ejecución.

Acción Write

Pre (s, write (va, val)) :=

context.os-accessible[va] = True $\wedge$ *s.aos-activity = running* $\wedge$

$\exists$ *(ma:madd), va-mapped-to-ma (s, va, ma)* $\wedge$

page-type (s.memory[ma]) = RW

Post (s, write (va, val), s') :=

$\exists$ *(ma:madd), va-mapped-to-ma (s, va, ma)* $\wedge$

$s \mathrel{\widehat{=}}_{memory:=memory'} s'$

,donde *memory' = (s.memory[ma] :=* {*page-content := RW (Some val), page-owned-by := OS (s.active-os)*}*)*

La precondición de esta acción es idéntica a la de la acción read. Lo único que hay que tener en cuenta es que la dirección virtual *va* es sobre la cual se quiere escribir.

La poscondición de esta acción pide que en el nuevo estado, en la memoria, la dirección de máquina asociada a *va* tenga una página cuyo contenido sea el valor que se quería escribir, siendo el guest OS activo su dueño.

3.3.7. Ejecución Válida de una Acción

La ejecución (válida) de una acción *act* en un estado válido *s*, la cual lleva al sistema a un estado *s'*, es representada por la relación $\hookrightarrow$.

$$\frac{\textit{valid-state(s)} \qquad \textit{Pre (s, act)} \qquad \textit{Post (s, act, s')}}{s \hookrightarrow_{act} s'}$$

La misma pide que cuando la precondición se mantenga, el estado (válido) cambie de manera que la poscondición de la acción quede establecida.

3.3.8. Invarianza de Estado Válido

Luego de la ejecución de una acción cualquiera el sistema debe estar asociado a un estado válido. Esto permite probar que este cumple con otras propiedades de interés, como por ejemplo isolation de guest OSs a nivel de ejecución de una acción (i.e. la ejecución de una acción en un guest OS no depende ni se ve afectada por el estado de los demás guest OSs) y isolation de guest OSs a nivel de trazas (o secuencia arbitraria de ejecuciones válidas). Por lo tanto, que la validez del estado se mantenga invariante luego de la ejecución de una acción, propiedad que llamaremos ***Invarianza de Estado Válido***, es gran importancia para el modelo.

Formalmente, esta propiedad queda de la siguiente manera:

Teorema: *invarianza_estado_valido: (act: action) (s s': state),*

$$\textit{valid-state(s)} \Rightarrow s \hookrightarrow_{act} s' \Rightarrow \text{valid-state(s')}$$

,donde *s'* representa el estado obtenido luego de la ejecución de la acción *act*.

En [10] se demuestra la válidez de esta propiedad.

Capítulo 4

Modelo Idelizado de Plataforma de Virtualización con Cache y TLB

En este capítulo se agregará al modelo la descripción de la Cache y de la TLB y se explicará que cambios produce dicha modificación.

4.1. Direccionamiento de la Memoria con Cache y TLB

Al modificar la especificación del modelo de la memoria agregándole Cache y TLB, el mismo queda como en la figura 4.1:

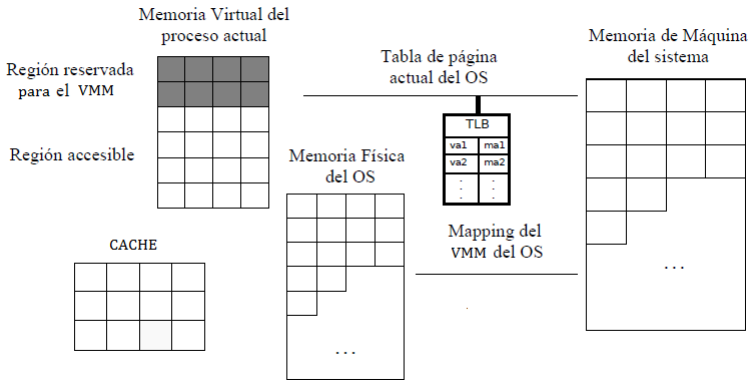

Figura 4.1: Esquema de direccionamiento de la memoria con Cache y TLB

Dichas modificaciones no cambian lo explicado en la sección 3.1, pero sí cambian la manera

en la que se accede a la información que se albergue en la memoria y la manera en que se buscan las traducciones de las direcciones virtuales.

4.2. Cache

La Cache agregada es del tipo VIVT. Esto significa que ahora cuando se quiera acceder a la información correspondiente a una dirección virtual, si la misma se halla en la Cache no será necesario traducirla. La misma será representada como un mapping con el siguiente tipo:

$$\textit{cache-struct} := \textit{vadd} \rightarrow \textit{page}$$

Respecto de su diseño, el tamaño máximo de la Cache está representado por el valor abstracto *max-cache*. De lo anterior se desprende que la Cache es un mapping acotado.

Dado que la Cache es representada como un mapping, los cuales son implementados utilizando listas, se optó por utilizar una función de correspondencia *Asociativa* y un algoritmo de reemplazo tipo *FIFO*.

Además, como el modelo idealizado supone que para toda dirección virtual que se encuentra en la Cache su dirección de maquina asociada se halla en la memoria, entonces se optó por utilizar una política de escritura *Write-Through*.

Cabe destacar que para evitar problemas de sinónimos y/o homónimos de direcciones en la Cache, la misma es vacíada cada vez que ocurre un cambio de contexto.

4.2.1. Operaciones de la Cache

- *f-cache-add: cache-struct* $\rightarrow$ *vadd* $\rightarrow$ *page* $\rightarrow$ *cache-struct*
  Función utilizada para agregar una nueva entrada en la Cache recibida como argumento. Diremos que una Cache c_2 es el resultado de agregar la entrada *(va, pg)* a una Cache c_1 sii

  $pg = c_2[va] \land \textit{cache-distinct}(c_2) \land \textit{map-size}\ c_2 \leq \textit{max-cache} \land$

  $\forall$ *(va':vadd) (pg':page)*, $va <> va' \Rightarrow pg' = c_2[va'] \Rightarrow pg' = c_1[va']$

  donde *cache-distinct* es un predicado que se cumple cuando la Cache que recibe no tiene entradas repetidas, $<>$ es el predicado *distintos* y *map-size* es una función que recibe un mapping y devuelve su tamaño.

Dicha propiedad la representaremos con el siguiente predicado:

cache-add *(c_1 c_2:cache-struct) (va:vadd) (pg:page): Prop :=*
Value pg = c_2[va] ∧ cache-distinct(c_2) ∧ map-size c_2 ≤ max-cache ∧
∀ (va':vadd) (pg':page), va <> va' ⇒ Value pg' = c_2[va'] ⇒ Value pg' = c_1[va']

- *f-cache-drop: cache-struct → vadd → cache-struct*
  Función utilizada para quitar entradas de la Cache recibida como argumento.

- *fix-cache-synonym: cache-struct → madd → cache-struct*
  Función utilizada para eliminar sinónimos de direcciones virtuales en la Cache recibida como argumento.

- *cache-flush: cache-struct → cache-struct*
  Función utilizada para vaciar la Cache recibida como argumento.

4.3. TLB

Agregar una TLB al modelo implica que cada vez que se quiera traducir una dirección virtual, primero se buscará en la TLB si esta se halla almacenada dicha traducción. La misma será representada como un mapping con el siguiente tipo:

tlb-struct := vadd → madd

Como se mencionó anteriormente, dado que la TLB es representada como un mapping, se optó por utilizar un algoritmo de reemplazo tipo *FIFO*. Además, a lo igual que la Cache, la TLB es un mapping acotado cuyo tamaño máximo es representado por el valor abstracto *max-tlb*. Para evitar problemas de sinónimos y/o homónimos de direcciones en la TLB, la misma es vacíada cada vez que ocurre un cambio de contexto.

4.3.1. Operaciones de la TLB

- *f-tlb-add: tlb-struct → vadd → madd → tlb-struct*
  Función utilizada para agregar una nueva entrada en la TLB recibida como argumento. Diremos que una TLB t_2 es el resultado de agregar la entrada *(va, ma)* a una TLB t_1 sii

ma = t_2*[va]* $\wedge$ tlb-distinct(t_2) $\wedge$ *map-size* $t_2 \leq$ *max-tlb* $\wedge$

$\forall$ *(va':vadd) (ma':page), va <> va'* $\Rightarrow$ *ma' =* t_2*[va'] ⇒ ma' =* t_1*[va']*

donde *tlb-distinct* es un predicado que se cumple cuando la TLB que recibe no tiene entradas repetidas. Esta propiedad la representaremos con el siguiente predicado:

tlb-add *(t1 t2:tlb-struct) (va:vadd) (ma:madd): Prop :=*
Value ma = t_2*[va]* $\wedge$ *tlb-distinct*(t_2) $\wedge$ *map-size* $t_2 \leq$ *max-tlb* $\wedge$
$\forall$ *(va':vadd) (ma':madd), va <> va'* $\Rightarrow$ *Value ma' =* t_2*[va'] ⇒ Value ma' =* t_1*[va']*

- *f-tlb-drop: tlb-struct → vadd → tlb-struct*
  Función utilizada para quitar entradas de la TLB recibida como argumento.

- *tlb-flush: tlb-struct → tlb-struct*
  Función utilizada para vaciar la TLB recibida como argumento.

4.4. Formalización del modelo

4.4.1. Estado

Al agregarse la Cache y la TLB, el estado es modelado de la siguiente manera:

state := {*active-os: os-ident,*
aos-exec-mode: exec-mode,
aos-activity: os-activity,
oss: oss-map,
hypervisor: hypervisor-map,
memory: system-memory,
cache: cache-struct,
tlb: tlb-struct}

dichos campos son como los definidos en la sección 3.3.3, agregando:

- **cache:** Representa la Cache del sistema.

- **tlb:** Representa la TLB del sistema.

4.4.2. Estado Válido

Al predicado *valid-state*, hay que agregarle las siguientes propiedades:

- **valid-cache:** El tamaño de la Cache no puede ser mayor a su tamaño máximo (*max-cache*) y la Cache no debe poseer entradas repetidas. Además, si una página *pg* está asociada a la dirección virtual *va* en la Cache, entonces *va* debe ser traducida a una dirección de máquina *ma* tal que *s.memory[ma] = pg* y *pg* sea una página de lectura/escritura.

- **valid-tlb:** El tamaño de la TLB no puede ser mayor a su tamaño máximo (*max-tlb*), la TLB no debe poseer entradas repetidas y para toda dirección virtual en la tlb, su dirección de máquina asociada en la antes mencionada debe coincidir con la dirección de máquina asociada a ellas en la tabla de paginación actual del guest OS que sea su dueño.

4.4.3. Semántica de las Acciones

Si bien no modifica los objetivos que las acciones persiguen, agregar Cache y TLB al modelo modifica, en algunos casos, sus comportamientos. A continuación analizaremos esto a partir de los ejemplos dados en la sección 3.2.6. Nuevamente, *s'* representa al estado obtenido luego de ejecutarse la acción en cuestión.

Acción Read

Pre (s, read (va)) :=

context.os-accessible[va] = True $\wedge$ *s.aos-activity = running* $\wedge$

$\exists$ *(ma:madd), va-mapped-to-ma (s, va, ma)* $\wedge$

page-type (s.memory[ma]) = RW

Post (s, read (va), s') :=

$\forall$ *(ma:madd) (pg:page), Some ma = va-mapped-to-ma-cache s va* $\Rightarrow$

Some pg = va-mapped-to-pg-cache s va $\Rightarrow$

cache-add s.cache (f-cache-add s.cache va pg) va pg $\wedge$

tlb-add s.tlb (f-tlb-add s.tlb va ma) va ma $\wedge$

$s \mathrel{\widehat{=}}_{cache:=cache',tlb:=tlb'} s'$

,donde *cache' = f-cache-add s.cache va pg*; y *tlb' = f-tlb-add s.tlb va ma.*

La precondición de esta acción no se modifica en lo absoluto. En cambio, la poscondición de esta acción ahora pide que si *va* está asociada a una dirección de máquina *ma* y *pg* es su página correspondiente, entonces hay que agregar a la Cache la entrada *(va, pg)* y hay que agregar a la TLB la entrada *(va, ma)*.

Acción Write

Pre (s, write (va, val)) :=

context.os-accessible[va] = True $\wedge$ *s.aos-activity = running* $\wedge$

$\exists$ *(ma:madd), va-mapped-to-ma (s, va, ma)* $\wedge$

page-type (s.memory[ma]) = RW

Post (s, write (va, val), s') :=

$\exists$ *(ma:madd), Some ma = va-mapped-to-ma-cache s va* $\wedge$

cache-add s.cache (f-cache-add (fix-cache-synonym s.cache ma) va pg) va pg $\wedge$

tlb-add s.tlb (f-tlb-add s.tlb va ma) va ma $\wedge$

$s \mathrel{\widehat{=}}_{memory:=memory',cache:=cache',tlb:=tlb'} s'$

,donde *pg* = {*page-content := RW (Some val), page-owned-by := OS (s.active-os)*}; *memory' = (s.memory[ma] := pg)*; *cache' = f-cache-add (fix-cache-synonym s.cache ma) va pg*; y *tlb' = f-tlb-add s.tlb va ma*.

La precondición de esta acción no se modifica en lo absoluto. En cambio, la poscondición de esta acción ahora también pide que antes de ir a la tabla de paginación en busca de la traducción de *va* primero chequear si la misma ya se encuentra en la TLB (*va-mapped-to-ma-cache*), que hay que agregar a la Cache la entrada *(va, pg)* (chequeando previamente que no haya sinónimos de direcciones en la misma) y que hay que agregar a la TLB la entrada *(va, ma)*.

4.4.4. Ejecución Válida de una Acción

Nuevamente, la ejecución (válida) de una acción *act* en un estado válido *s*, la cual lleva al sistema a un estado *s'*, es representada por la relación $\hookrightarrow$.

$$\frac{\textit{valid-state(s)} \qquad \textit{Pre (s, act)} \qquad \textit{Post (s, act, s')}}{s \hookrightarrow_{act} s'}$$

La misma pide que cuando la precondición se mantenga, el estado (válido) cambie de manera que la poscondición de la acción quede establecida.

4.4.5. Invarianza de Estado Válido

Las acciones que conteplan el uso de Cache y TLB respetan la propiedad de ***Invarianza de Estado Válido***. Sin embargo, la demostración de lo anterior, la cual puede ser consultada en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php` (en el modulo COQ **Exec_invariant.v**), no fue parte de este trabajo.

Capítulo 5

Especificación Ejecutable

En algunas partes de este capítulo se hará mención a algunas funciones pero no se mostrará su implementación. Sin embargo, dichas implementaciones pueden ser consultadas en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

5.1. Implementación de las operaciones de la Cache y la TLB

5.1.1. Operaciones de la Cache

Como ya se dijo en la sección 4.2, la Cache se define como un mapping con el siguiente tipo:

cache-struct := mapping vadd page
cache: cache-struct

Respecto de sus operaciones, como *fix-cache-synonym* y *cache-flush* fueron provistas por el modelo solo se analizarán *f-cache-add* y *f-cache-drop*, las cuales fueron implementadas en este trabajo.

f-cache-drop

La operación *f-cache-drop* se define simplemente como un drop de un mapping. Esto es,

f-cache-drop (c:cache-struct) (va:vadd) : cache-struct := map-drop c va

f-cache-add

La operación *f-cache-add* no es tan sencilla dado que la manera de agregar un elemento a la Cache variará según el estado de la anterior y según si la dirección virtual a modificar está o no en la Cache.

f-cache-add (c:cache-struct) (va:vadd) (pg:page) : cache-struct :=
if *map-valid-index c va*
then *map-add c va pg*
else if *is-full-cache c*
then replace−page c va pg
else add c va pg

Si la dirección virtual *va* es un índice de la Cache *c*, hecho que se chequea con la función *map-valid-index*, entonces simplemente se hace un *add* de mappings dado que esta función actualiza los valores si el elemento en cuestión (en este caso *va*) se encuentra en el mapping. Si no lo es, entonces se chequea, utilizando la función *is-full-cache*[1], si la Cache está llena o no. Si lo está, entonces se debe aplicar el algoritmo de remplazo de páginas (*replace-page*). Si no, simplemente se agrega la entrada *(va, pg)* en *c*. Cabe destacar que en este último caso se utiliza la función *add* y no la función *map-add* dado que la primera agrega el elemento en *c* de tal manera de que el algoritmo de remplazo se pueda aplicar fácilmente.

Las funciones *replace-page* y *add* fueron implementadas de la siguiente manera:

add := fifo-add
replace-page := fifo-replace

donde,

fifo-drop (c:cache-struct): cache-struct :=
match *c* ***with***
$|$ *map-empty* $\Rightarrow$ *map-empty*
$|$ *x :: xs* $\Rightarrow$ *xs*
end

[1]Es fácil ver que si la Cache *c* está llena entonces *map-size c = max-cache*.

```
fifo-add (c:cache-struct) (va:vadd) (pg:page): cache-struct :=
match c with
        |map-empty ⇒ (Item va pg) :: map-empty
        |x :: xs ⇒ x :: fifo-add xs va pg
end
```

fifo-replace (c:cache-struct) (va:vadd) (pg:page): cache-struct := fifo-add (fifo-drop c) va pg

5.1.2. Operaciones de la TLB

Las operaciones de la TLB son análogas a las operaciones de la Cache. Simplemente, hay que cambiar en los tipos de las operaciones de la Cache *cache-struct* por *tlb-struct* y *page* por *madd* y listo, las operaciones resultantes a dichos cambios son las operaciones de la TLB.

5.2. Implementación de las Acciones

A continuación se explicará de que manera se implementaron algunas de las acciones del modelo. El total de las mismas puede ser consultado en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

Cabe destacar que en las acciones *page-pin-untrusted* y *page-pin-trusted* hubo que agregar como argumento *ma:madd* el cual representa la dirección libre a utilizar y chequeos que aseguren que *ma* efectivamente está libre.

Acción Read

Supongamos que queremos leer la página asociada a la dirección virtual *va* en el estado *s*. Si dicha dirección se halla en (i.e. es un índice de) la Cache, entonces se puede acceder directamente a la página deseada. Luego, solo queda fijarse si es necesario actualizar la TLB. Si no es necesario, entonces el estado no sufre modificaciones. Si no existe la traducción de *va* en la TLB, entonces debemos proceder a buscar dicha traducción en la tabla de paginación y luego agregarla a la TLB. En cambio, si *va* no está en la Cache debemos proceder a buscar su traducción. Si la misma se halla en la TLB, ya se puede acceder a la página deseada. Luego, simplemente hay que agregar a la Cache la entrada *(va, página-asociada-a-va)*. Si no, entonces debemos proceder a buscar dicha traducción en la tabla de paginación. Una vez hallada la traducción ya se puede acceder

a la página deseada. Por lo tanto, solo queda agregar a la TLB la entrada *(va, traduccion-va)* y agregar a la Cache la entrada *(va, página-asociada-a-va)*. En ambos casos, cuando la búsqueda en la tabla de paginación falle dicho fallo se considera un error, por lo que en dicho caso el estado no sufrirá modificaciones.

Formalmente, la acción *read* se implementa de la siguiente manera:

```
read-action (s:state) (va:vadd) : state :=
if map-valid-index s.cache va
then if map-valid-index s.tlb va
     then s
     else match va-mapped-to-ma-cache s va with
          | None ⇒ s
          | Some ma ⇒ update-state-tlb s (f-tlb-add s.tlb va ma)
          end
else if map-valid-index s.tlb va
     then match s.tlb[va] with
          | Error - ⇒ s
          | Value ma ⇒ match s.memory[ma] with
                       | Error - ⇒ s
                       | Value pg ⇒ update-state-cache s (f-cache-add s.cache va pg)
                       end
          end
     else match va-mapped-to-ma-cache s va with
          | None ⇒ s
          | Some ma ⇒ let new-s := update-state-tlb s (f-tlb-add s.tlb va ma) in
                      match s.memory[ma] with
                      | Error - ⇒ s
                      | Value pg ⇒ update-state-cache new-s (f-cache-add s.cache va pg)
                      end
          end
```

donde,

- *va-mapped-to-ma-cache s va* busca la dirección de máquina asociada a (i.e. la traducción

de) la dirección virtual *va* en el estado *s*.

- *update-state-cache s c* cambia el valor del campo *s.cache* por el valor *c* (s $\widehat{=}_{cache}$*update-state-cache s c*).
- *update-state-tlb s t* cambia el valor del campo *s.tlb* por el valor *t* (s $\widehat{=}_{tlb}$*update-state-tlb s t*).

Acción Write

Supongamos que queremos modificar la página asociada a la dirección virtual *va* con el valor *val* en el estado *s*. Si dicha dirección se halla en (i.e. es un índice de) la Cache, entonces, dado que el tipo de política de escritura utilizado es *Write-Through*, debemos actualizar el valor asociado a *va* con *val* tanto en la Cache como en la memoria. Si no, debemos actualizar solamente el valor antes mencionado con *val* en la memoria y agregar en la Cache la entrada *(va*, {*page-content := RW (Some val), page-owned-by := va-owner*}*)*, donde *va-owner* representa al guest OS dueño de la dirección *va*. Además, si en la TLB no se encuentra una traducción asociada a la dirección virtual *va*, entonces debemos buscar en la tabla de paginación actual del guest OS que este corriendo dicha traducción y agregarla a la TLB. Si dicha traducción ya se encuentra en la TLB, entonces la TLB no sufre modificaciones.

Formalmente, la acción *write* se implementa de la siguiente manera:

```
write-action (s:state) (va:vadd) (val:value) : state :=
match s.cache[va], s.tlb[va] with
  | Value old-pg, Value ma
      => match s.memory[ma] with
           | Value - ⇒ update-cache-mem s va (RW (Some val)) old-pg
           | Error - ⇒ s
         end
  | Value old-pg, Error -
      => match va-mapped-to-ma-cache s va with
           | None ⇒ s
           | Some ma ⇒ match s.memory[ma] with
                       | Value - ⇒ let new-state := update-state-tlb s (f-tlb-add s.tlb va ma)
                                   in update-cache-mem new-state va (RW (Some val)) old-pg
```

```
                    | Error - ⇒ s
                        end
            end
| Error -, Value ma
    => match s.memory[ma] with
            | Value - ⇒ update-cache-mem s va (RW (Some val)) old-pg
            | Error - ⇒ s
        end
| Error -, Error -
    => match va-mapped-to-ma-cache s va with
            | None ⇒ s
            | Some ma ⇒ match s.memory[ma] with
                    | Value - ⇒ let new-state := update-state-tlb s (f-tlb-add s.tlb va ma)
                                in update-cache-mem new-state va (RW (Some val)) old-pg
                    | Error - ⇒ s
                        end
            end
end
```

donde,

- *va-mapped-to-ma-cache s va* busca la dirección de máquina asociada a (i.e. la traducción de) la dirección virtual *va* en el estado *s*.

- *update-state-cache s c* cambia el valor del campo *s.cache* por el valor *c* (s $\widehat{=}_{cache}$*update-state-cache s c*).

- *update-state-tlb s t* cambia el valor del campo *s.tlb* por el valor *t* (s $\widehat{=}_{tlb}$*update-state-tlb s t*).

- *update-cache-mem (s:state) (va:vadd) (ma:madd) (c:content) (old-pg:page): state :=*
  let *new-pg* := {*page-content := c, page-owned-by := old-pg.page-owned-by*} **in**
  let *new-state := update-state-cache s (f-cache-add (fix-cache-synonym s s.cache ma) va new-pg)* **in** *update-aos-va-cont new-state va new-pg*

- *update-aos-va-cont s va pg* actualiza el contenido de la página asociada a la dirección virtual *va* con el contenido de la página *pg* en la memoria (i.e. s.memory) solo si el dueño de *pg* es el mismo que el de la página asociada a *va*. En caso de error, esta función devuelve *s* sin modificaciones.

5.3. Verificación de Corrección

5.3.1. Formato de las demostraciones

El planteo de las propiedades a demostrar tendrá un formato similar al utilizado en el cálculo de secuentes:

$$\frac{\begin{array}{l} H_1\text{: } expresion_1 \\ H_2\text{: } expresion_2 \\ . \\ . \\ H_n\text{: } expresion_n \end{array}}{Goal}$$

En el mismo, $\{H_1, H_2, ..., H_n\}$ será el conjunto de las hipótesis disponibles para llevar adelante la prueba y *Goal* será la propiedad que queremos demostrar.

Para desarrollar la demostración, primero se dará una explicación textual de los cambios a realizar en un determinado paso de la misma e inmediatamente a continuación de dicha explicación se mostrará como queda el "secuente" modificado con dichos cambios.

Por lo general, salvo que se mencione explícitamente a que expresión o hipótesis se haga referencia, cuando se hable de la expresión antes mencionada estaremos haciendo referencia al *Goal* del "secuente" anterior.

Ejemplo:

$$\frac{H_1\text{: } foo = 2}{foo + 2 = 4}$$

Dada la hipótesis H_1, la expresión anterior es equivalente a la expresión,

$$\frac{H_1\text{: } foo = 2}{2 + 2 = 4}$$

Esta última expresión, luego de realizar la suma, es válida por reflexividad.

5.3.2. Pruebas de Corrección de las Operaciones de la Cache y la TLB

Como ya se explicó en la sección 5.1.2 las operaciones de la Cache son análogas a las operaciones de la TLB. Esto mismo ocurre con las demostraciones de sus pruebas de corrección. Por lo tanto, solo nos enfocaremos en la corrección de las operaciones de la Cache. Vale recordar que tanto las operaciones de la TLB como sus pruebas de corrección se pueden consultar en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`, en el modulo COQ **Cache.v**.

f-cache-drop

Es trivial ver que al eliminar una entrada de la Cache la misma sigue siendo una Cache válida (i.e. que el predicado *valid-cache* se mantiene). Por lo tanto, no es necesario explayarse en esta prueba de corrección.

f-cache-add

f-cache-add (c:cache-struct) (va:vadd) (pg:page) : cache-struct :=
if *map-valid-index c va*
then *map-add c va pg*
else if *is-full-cache c*
then *replace-page c va pg*
else *add c va pg*

En la sección 4.2.1 vimos que esta operación debe cumplir con el predicado *cache-add*. Entonces, para probar la corrección de esta operación debemos demostrar el siguiente teorema:

Teorema *f-cache-add-correct : forall (old-cache:cache) (va:vadd) (pg:page),*
valid-cache old-cache ⇒ cache-add old-cache (f-cache-add old-cache va pg) va pg.

Primero hay que suponer el antecedente. Luego, como *cache-add* son varias conjunciones, hay que analizar cada uno de sus átomos por separado.

A continuación solo se dará a modo ejemplo la demostración de uno de los átomos de dicha proposición. Las restantes demostraciones pueden ser consultadas en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`, en el modulo COQ **Cache.v**.

map-size (f-cache-add old-cache va pg) $\leq$ ***max-cache***

Llamando H_1 a la hipótesis supuesta hay que demostrar,

$$\frac{H_1\text{: } \textit{valid-cache old-cache}}{\textit{map-size (f-cache-add old-cache va pg)} \leq \textit{max-cache}}$$

Dadas las condiciones *map-valid-index old-cache va* y *is-full-cache old-cache* de la operación *f-cache-add*, analizamos los siguientes 3 casos relevantes:

I) *map-valid-index old-cache va = True* $\wedge$ *is-full-cache old-cache = True*

II) *map-valid-index old-cache va = False* $\wedge$ *is-full-cache old-cache = True*

III) *map-valid-index old-cache va = False* $\wedge$ *is-full-cache old-cache = False*

- En el ***primer caso*** la expresión a demostrar queda,

$$\frac{\begin{array}{l}H_1\text{: } \textit{valid-cache old-cache}\\ H_2\text{: } \textit{map-valid-index old-cache va = True}\\ H_3\text{: } \textit{is-full-cache old-cache = True}\end{array}}{\textit{map-size (map-add old-cache va pg)} \leq \textit{max-cache}}$$

  Por el lema *map_add_valid_size*[2], si agregamos una entrada a la Cache cuyo índice ya está en la misma, entonces la Cache no cambiará su tamaño. Por lo tanto, la expresión antes mencionada es equivalente a la expresión,

$$\frac{\begin{array}{l}H_1\text{: } \textit{valid-cache old-cache}\\ H_2\text{: } \textit{map-valid-index old-cache va = True}\\ H_3\text{: } \textit{is-full-cache old-cache = True}\end{array}}{\textit{map-size old-cache} \leq \textit{max-cache}}$$

  Luego, por hipótesis H_1, dicha expresión es válida.

- En el ***segundo caso*** la expresión a demostrar queda,

[2]Ver modulo COQ **Maps_proof.v** en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache old-cache} \\ H_2\text{: } \textit{map-valid-index old-cache va} = \textit{False} \\ H_3\text{: } \textit{is-full-cache old-cache} = \textit{True} \end{array}}{\textit{map-size (replace-page old-cache va pg)} \leq \textit{max-cache}}$$

Por la definición de *replace-page*, esta expresión es equivalente a la expresión,

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache old-cache} \\ H_2\text{: } \textit{map-valid-index old-cache va} = \textit{False} \\ H_3\text{: } \textit{is-full-cache old-cache} = \textit{True} \end{array}}{\textit{map-size (fifo-add (fifo-drop old-cache) va pg)} \leq \textit{max-cache}}$$

Ahora, procedemos a realizar inducción estructural sobre la estructura de *old-cache*.

En el ***Caso Base*** tenemos que probar,

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache map-empty} \\ H_2\text{: } \textit{map-valid-index map-empty va} = \textit{False} \\ H_3\text{: } \textit{is-full-cache map-empty} = \textit{True} \end{array}}{\textit{map-size (fifo-add (fifo-drop map-empty) va pg)} \leq \textit{max-cache}}$$

Ahora bien, por hipótesis H_3 tenemos que,

$$0 = \textit{map-size map-empty} = \textit{max-cache}$$

; pero en el modelo se supone $\textit{max-cache} > 0$. Por lo tanto, este caso es ***absurdo***.

En el ***Caso Inductivo*** tenemos que probar,

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache (c :: cs)} \\ H_2\text{: } \textit{map-valid-index (c :: cs) va} = \textit{False} \\ H_3\text{: } \textit{is-full-cache (c :: cs)} = \textit{True} \\ \textit{HI: map-size (fifo-add (fifo-drop cs) va pg)} \leq \textit{max-cache} \end{array}}{\textit{map-size (fifo-add (fifo-drop (c :: cs)) va pg)} \leq \textit{max-cache}}$$

Si *indice c = va* estaríamos en un caso *absurdo* dado que la hipótesis H_2 quedaría *True = False*. Por lo tanto, *indice c <> va* (hecho que agregamos como hipótesis) y la hipótesis H_2 queda,

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache (c :: cs)} \\ H_2\text{: } \textit{map-valid-index cs va = False} \\ H_3\text{: } \textit{is-full-cache (c :: cs) = True} \\ \textit{HI: map-size (fifo-add (fifo-drop cs) va pg)} \leq \textit{max-cache} \\ H_4\text{: } \textit{indice c <> va} \end{array}}{\textit{map-size (fifo-add (fifo-drop (c :: cs)) va pg)} \leq \textit{max-cache}}$$

Por el lema *add-idem-map-add*[3], dado que *map-valid-index old-cache va = False*, el tamaño de la Cache al agregarse una entrada con índice *va* es igual al tamaño de la Cache incrementado en uno. De esto se desprende que la expresión a demostrar es equivalente a la expresión,

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache (c :: cs)} \\ H_2\text{: } \textit{map-valid-index cs va = False} \\ H_3\text{: } \textit{is-full-cache (c :: cs) = True} \\ \textit{HI: map-size (fifo-add (fifo-drop cs) va pg)} \leq \textit{max-cache} \\ H_4\text{: } \textit{indice c <> va} \end{array}}{\textit{map-size (fifo-drop (c :: cs))} + 1 \leq \textit{max-cache}}$$

Además, como ya se mostró en el *Caso Base*, la hipótesis H_3 significa,

$$\textit{map-size (c :: cs) = max-cache}$$

Por lo tanto, agregamos dicha propiedad a nuestro conjunto de hipótesis.

[3]Ver modulo COQ **Cache.v** en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache (c :: cs)} \\ H_2\text{: } \textit{map-valid-index cs va} = \textit{False} \\ H_3\text{: } \textit{is-full-cache (c :: cs)} = \textit{True} \\ \textit{HI: map-size (fifo-add (fifo-drop cs) va pg)} \leq \textit{max-cache} \\ H_4\text{: } \textit{indice c} <> \textit{va} \\ H_5\text{: } \textit{map-size (c :: cs)} = \textit{max-cache} \end{array}}{\textit{map-size (fifo-drop (c :: cs)) + 1} \leq \textit{max-cache}}$$

Luego, por hipótesis H_5, la expresión a demostrar es equivalente a la expresión,

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache (c :: cs)} \\ H_2\text{: } \textit{map-valid-index cs va} = \textit{False} \\ H_3\text{: } \textit{is-full-cache (c :: cs)} = \textit{True} \\ \textit{HI: map-size (fifo-add (fifo-drop cs) va pg)} \leq \textit{max-cache} \\ H_4\text{: } \textit{indice c} <> \textit{va} \\ H_5\text{: } \textit{map-size (c :: cs)} = \textit{max-cache} \end{array}}{\textit{map-size (fifo-drop (c :: cs)) + 1} \leq \textit{map-size (c :: cs)}}$$

Esta expresión, por la definición de *fifo-drop* y de *map-size*, es equivalente a la expresión,

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache (c :: cs)} \\ H_2\text{: } \textit{map-valid-index cs va} = \textit{False} \\ H_3\text{: } \textit{is-full-cache (c :: cs)} = \textit{True} \\ \textit{HI: map-size (fifo-add (fifo-drop cs) va pg)} \leq \textit{max-cache} \\ H_4\text{: } \textit{indice c} <> \textit{va} \\ H_5\text{: } \textit{map-size (c :: cs)} = \textit{max-cache} \end{array}}{\textit{map-size cs + 1} \leq \textit{map-size cs + 1}}$$

Esta última expresión es válida por reflexividad.

- En el ***tercer caso*** la expresión a demostrar queda,

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache old-cache} \\ H_2\text{: } \textit{map-valid-index old-cache va} = \textit{False} \\ H_3\text{: } \textit{is-full-cache old-cache} = \textit{False} \end{array}}{\textit{map-size (add old-cache va pg)} \leq \textit{max-cache}}$$

Por el lema *add-idem-map-add*[4], dado que *map-valid-index old-cache va = False*, el tamaño de la Cache al agregarse una entrada con índice *va* es igual al tamaño de la Cache incrementado en uno. Por lo tanto, la expresión anterior es equivalente a la expresión,

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache old-cache} \\ H_2\text{: } \textit{map-valid-index old-cache va} = \textit{False} \\ H_3\text{: } \textit{is-full-cache old-cache} = \textit{False} \end{array}}{\textit{map-size old-cache} + 1 \leq \textit{max-cache}}$$

Ahora bien, como la hipótesis H_1 es válida sabemos que *map-size old-cache* $\leq$ *max-cache*. Además, por la hipótesis H_3 sabemos que *map-size old-cache* <> *max-cache*. Por lo tanto, tenemos que *map-size old-cache* $<$ *max-cache*.

Dicha desigualdad es equivalente a la desigualdad *map-size old-cache + 1* $\leq$ *max-cache*. Luego, agregamos la propiedad anterior a nuestro conjunto de hipótesis,

$$\frac{\begin{array}{l} H_1\text{: } \textit{valid-cache old-cache} \\ H_2\text{: } \textit{map-valid-index old-cache va} = \textit{False} \\ H_3\text{: } \textit{is-full-cache old-cache} = \textit{False} \\ H_4\text{: } \textit{map-size old-cache} + 1 \leq \textit{max-cache} \end{array}}{\textit{map-size old-cache} + 1 \leq \textit{max-cache}}$$

Entonces, por la hipótesis H_4, la expresión a demostrar es válida.

QED

5.3.3. Pruebas de Corrección de las Acciones

Diremos que una acción es correcta si su ejecución es válida (ver sección 3.3.7). Para probar que la ejecución de una acción es válida debemos demostrar para cada acción un teorema con la siguiente forma:

Teorema act-correcta: ∀ (s:state) (act:action),
valid-state s ⇒ *Pre (s, act)* ⇒ *Post (s, act, f-act s)*

[4]Ver modulo COQ **Cache.v** en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

, donde *f-act* es la implementación de la acción *act* y *f-act s* representa al estado resultante luego de ejecutar la acción *act* en el estado *s*.

A continuación se dará solamente la prueba de corrección de la acción *write*. El resto de las pruebas puede ser consultado en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php` en el modulo COQ **Actions_impl_cache.v**.

Prueba de Corrección de la Acción Write

Teorema write-correcta: ∀ (s:state) (va:vadd) (val:value),

valid-state s ⇒ *Pre (s, write (va, val))* ⇒ *Post (s, write (va, val), write-action s va val)*

Primero hay que suponer los antecedentes. Entonces hay que probar,

$$\frac{\begin{array}{l} H_1\text{: } valid\text{-}state\ s \\ H_2\text{: } Pre\ (s,\ write\ (va,\ val)) \end{array}}{Post\ (s,\ write\ (va,\ val),\ write\text{-}action\ s\ va\ val)}$$

Esta expresión, a partir de la definición de la poscondición de *write* (ver sección 4.4.3), es equivalente a la expresión,

$$\frac{\begin{array}{l} H_1\text{: } valid\text{-}state\ s \\ H_2\text{: } Pre\ (s,\ write\ (va,\ val)) \\ H_3\text{: } pg = \{page\text{-}content := RW\ (Some\ val),\ page\text{-}owned\text{-}by := OS\ (s.active\text{-}os)\} \\ H_4\text{: } cache' = f\text{-}cache\text{-}add\ (fix\text{-}cache\text{-}synonym\ s.cache\ ma)\ va\ pg \\ H_5\text{: } tlb' = f\text{-}tlb\text{-}add\ s.tlb\ va\ ma \\ H_6\text{: } memory' = (update\text{-}aos\text{-}va\text{-}cont\ s\ va\ pg).memory \end{array}}{\begin{array}{l} \exists\ (ma\text{:}madd),\ Some\ ma = va\text{-}mapped\text{-}to\text{-}ma\text{-}cache\ s\ va\ \wedge \\ cache\text{-}add\ (fix\text{-}cache\text{-}synonym\ s.cache\ ma)\ cache'\ va\ pg\ \wedge \\ tlb\text{-}add\ s.tlb\ (f\text{-}tlb\text{-}add\ s.tlb\ va\ ma)\ va\ ma\ \wedge \\ s \mathrel{\widehat{=}}_{memory:=memory',cache:=cache',tlb:=tlb'} write\text{-}action\ s\ va\ val \end{array}}$$

; donde la dirección *ma* utilizada en las hipótesis H_4, H_5 y H_6 es la cuantificada existencialmente en el goal.

La precondición de la acción write nos dice que existe una dirección de máquina, llamémosla *ma0*, la cual está asociada a la dirección virtual *va*. Por lo tanto podemos utilizar *ma0* como testigo de la cuantificación existencial y agregar el hecho de que está asociada a *va* a nuestro conjunto de hipótesis.

H_1: *valid-state s*

H_2: *Pre (s, write (va, val))*

H_3: *pg = {page-content := RW (Some val), page-owned-by := OS (s.active-os)}*

H_4: *cache' = f-cache-add (fix-cache-synonym s.cache ma0) va pg*

H_5: *tlb' = f-tlb-add s.tlb va ma0*

H_6: *memory' = (update-aos-va-cont s va pg).memory*

H_7: *Some ma0 = va-mapped-to-ma-cache s va*

Some ma0 = va-mapped-to-ma-cache s va $\wedge$

cache-add (fix-cache-synonym s.cache ma) cache' va pg $\wedge$

tlb-add s.tlb (f-tlb-add s.tlb va ma0) va ma0 $\wedge$

$s \mathrel{\widehat{=}}_{memory:=memory',cache:=cache',tlb:=tlb'}$ *write-action s va val*

Dado que la expresión a demostrar es una conjunción, hay que analizar cada uno de sus átomos por separado.

Some ma0 = va-mapped-to-ma-cache s va

Esta expresión es válida por la hipótesis H_7.

cache-add (fix-cache-synonym s.cache ma) cache' va pg

Esta expresión, como vimos en la sección 5.3.2 tomando *(fix-cache-synonym s.cache ma)* como *old-cache*, es válida.

tlb-add s.tlb (f-tlb-add s.tlb va ma) va ma

Esta expresión, como vimos en la sección 5.3.2, es válida.

$s \mathrel{\widehat{=}}_{memory:=memory',cache:=cache',tlb:=tlb'}$ ***write-action s va val***

Esta expresión dice que, salvo en los casos de las componentes *memory*, *cache* y *tlb*, las componentes del estado no varían en el nuevo estado. Por lo tanto, esta expresión es válida solo si las siguientes 8 igualdades son válidas:

1. *s.active-os = (write-action s va val).active-os*

2. *s.aos-exec-mode = (write-action s va val).aos-exec-mode*

3. *s.aos-activity = (write-action s va val).aos-activity*

4. *s.oss = (write-action s va val).oss*

5. *s.hypervisor = (write-action s va val).hypervisor*

6. *(write-action s va val).memory = memory'*

7. *(write-action s va val).cache = cache'*

8. *(write-action s va val).tlb = tlb'*

Si analizamos la definición de *write-action* (sección 5.2) se ve a simple vista que el estado resultante de aplicar dicha acción solo puede sufrir modificaciones en sus campos *memory*, *cache* y *tlb*. Por lo tanto, las 5 primeras igualdades son válidas. Además, como las operaciones de la Cache son análogas a las de la TLB, la misma analogía ocurre con las demostraciones de las 2 últimas igualdades. Luego, solamente queda probar que,

H_1: *valid-state s*
H_2: *Pre (s, write (va, val))*
H_3: *pg = {page-content := RW (Some val), page-owned-by := OS (s.active-os)}*
H_4: *cache' = f-cache-add (fix-cache-synonym s.cache ma0) va pg*
H_5: *memory' = (update-aos-va-cont s va pg).memory*
H_6: *Some ma0 = va-mapped-to-ma-cache s va*

(write-action s va val).memory = memory' ∧
(write-action s va val).cache = cache'

Por las hipótesis H_4 y H_5 la expresión a demostrar es equivalente a la expresión,

H_1: *valid-state s*
H_2: *Pre (s, write (va, val))*
H_3: *pg = {page-content := RW (Some val), page-owned-by := OS (s.active-os)}*
H_6: *Some ma0 = va-mapped-to-ma-cache s va*

(write-action s va val).memory = (update-aos-va-cont s va pg).memory ∧
(write-action s va val).cache = f-cache-add (fix-cache-synonym s.cache ma0) va pg

Notar que las hipótesis H_4 y H_5 fueron quitadas de nuestro conjunto de hipótesis. Esto es así dado que a partir de el último cambio realizado en el goal dichas hipótesis no serán necesarias en el resto de la prueba.

La acción *write-action* presenta los siguientes 2 casos relevantes:

I) *map-valid-index s.cache va = True* $\wedge$ *map-valid-index s.tlb va = True* $\wedge$
map-valid-index s.memory ma0 = True

II) *map-valid-index s.cache va = True* $\wedge$ *map-valid-index s.tlb va = False* $\wedge$
map-valid-index s.memory ma0 = True

Como la demostraciones de ambos casos se desarrollan de una manera muy similar, solo se analizará el primer caso. El resto de la prueba ser consultado en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`, en el modulo COQ **Actions_impl_cache.v**.

En el ***primer caso*** esta acción queda,

update-cache-mem s va (RW (Some val)) old-pg

; donde *s.cache[va] = Value old-pg*, *s.tlb[va] = Value ma0* y *s.memory[ma] = Value old-pg.*

Esto significa que la expresión a demostrar queda,

H_1: *valid-state s*
H_2: *Pre (s, write (va, val))*
H_3: *pg* = {*page-content := RW (Some val), page-owned-by := OS (s.active-os)*}
H_6: *Some ma0 = va-mapped-to-ma-cache s va*
H_7: *map-valid-index s.cache va = True*
H_8: *map-valid-index s.tlb va = True*
H_9: *map-valid-index s.memory ma0 = True*

(update-cache-mem s va ma0 (RW (Some val)) old-pg).memory =
(update-aos-va-cont s va pg).memory $\wedge$
(update-cache-mem s va ma0 (RW (Some val)) old-pg).cache =
f-cache-add (fix-cache-synonym s.cache ma0) va pg

Es fácil ver que como el guest OS que este corriendo es dueño de *va* y *old-pg* es la página asociada a dicha dirección, entonces dicho guest OS es dueño de *old-pg*. Esto significa que, *old-pg.page-owned-by = OS (s.active-os)*, hecho que agregaremos a nuestro conjunto de hipótesis.

H_1: *valid-state s*
H_2: *Pre (s, write (va, val))*
H_3: *pg = {page-content := RW (Some val), page-owned-by := OS (s.active-os)}*
H_6: *Some ma0 = va-mapped-to-ma-cache s va*
H_7: *map-valid-index s.cache va = True*
H_8: *map-valid-index s.tlb va = True*
H_9: *map-valid-index s.memory ma0 = True*
H_{10}: *old-pg.page-owned-by = OS (s.active-os)*

(update-cache-mem s va ma0 (RW (Some val)) old-pg).memory =
(update-aos-va-cont s va pg).memory ∧
(update-cache-mem s va ma0 (RW (Some val)) old-pg).cache =
f-cache-add (fix-cache-synonym s.cache ma0) va pg

Ahora, si analizamos la definición de *update-cache-mem* la misma tiene 2 expresiones **let**. Entonces, podemos agregar dichas expresiones como los siguientes 2 axiomas a nuestro conjunto de hipótesis y después utilizar la definición de *update-cache-mem* para avanzar en la prueba,

new-pg = {page-content := RW (Some val), page-owned-by := old-pg.page-owned-by}
new-state = update-state-cache s (f-cache-add (fix-cache-synonym s s.cache ma) va new-pg)

Notése que simplemente se reemplazaron las asignaciones principales por iguales. Luego, por definición de *update-cache-mem*, la expresión a demostrar queda,

H_1: *valid-state s*
H_2: *Pre (s, write (va, val))*
H_3: *pg = {page-content := RW (Some val), page-owned-by := OS (s.active-os)}*
H_6: *Some ma0 = va-mapped-to-ma-cache s va*
H_7: *map-valid-index s.cache va = True*
H_8: *map-valid-index s.tlb va = True*
H_9: *map-valid-index s.memory ma0 = True*
H_{10}: *old-pg.page-owned-by = OS (s.active-os)*

H_{11}: new-pg = {page-content := RW (Some val), page-owned-by := old-pg.page-owned-by}
H_{12}: new-state = update-state-cache s (f-cache-add (fix-cache-synonym s.cache ma0) va new-pg)

(update-aos-va-cont new-state va new-pg).memory = (update-aos-va-cont s va pg).memory ∧
(update-aos-va-cont new-state va new-pg).cache =
f-cache-add (fix-cache-synonym s.cache ma0) va pg

Ahora, como la expresión a demostrar son varias conjunciones, analizamos cada uno de sus átomos por separado.

Primer Átomo

H_1: valid-state s
H_2: Pre (s, write (va, val))
H_3: pg = {page-content := RW (Some val), page-owned-by := OS (s.active-os)}
H_6: Some ma0 = va-mapped-to-ma-cache s va *H_7: map-valid-index s.cache va = True*
H_8: map-valid-index s.tlb va = True
H_9: map-valid-index s.memory ma0 = True
H_{10}: old-pg.page-owned-by = OS (s.active-os)
H_{11}: new-pg = {page-content := RW (Some val), page-owned-by := old-pg.page-owned-by}
H_{12}: new-state = update-state-cache s (f-cache-add (fix-cache-synonym s.cache ma0) va new-pg)

(update-aos-va-cont new-state va new-pg).memory = (update-aos-va-cont s va pg).memory

Por hipótesis H_{10}, es fácil ver que *new-pg = pg*. Esto significa que la expresión a demostrar es equivalente a la expresión,

H_1: valid-state s
H_2: Pre (s, write (va, val))
H_3: pg = {page-content := RW (Some val), page-owned-by := OS (s.active-os)}
H_6: Some ma0 = va-mapped-to-ma-cache s va
H_7: map-valid-index s.cache va = True

H_8: *map-valid-index s.tlb va = True*
H_9: *map-valid-index s.memory ma0 = True*
H_{10}: *old-pg.page-owned-by = OS (s.active-os)*
H_{11}: *new-pg = {page-content := RW (Some val), page-owned-by := old-pg.page-owned-by}*
H_{12}: *new-state = update-state-cache s (f-cache-add (fix-cache-synonym s.cache ma0) va new-pg)*

(update-aos-va-cont new-state va pg).memory = (update-aos-va-cont s va pg).memory

A partir de la hipótesis H_{12} es fácil ver que *new-state.memory = s.memory*, dado que en *new-state* se modifica solamente el campo *cache* de *s*. Esto significa que si en sendos campos *memory* modificamos las páginas asociadas a la misma dirección con el mismo valor entonces dicha igualdad se seguirá manteniendo. Por lo tanto, la expresión a demostrar es válida.

Segundo Átomo

H_1: *valid-state s*
H_2: *Pre (s, write (va, val))*
H_3: *pg = {page-content := RW (Some val), page-owned-by := OS (s.active-os)}*
H_6: *Some ma0 = va-mapped-to-ma-cache s va*
H_7: *map-valid-index s.cache va = True*
H_8: *map-valid-index s.tlb va = True*
H_9: *map-valid-index s.memory ma0 = True*
H_{10}: *old-pg.page-owned-by = OS (s.active-os)*
H_{11}: *new-pg = {page-content := RW (Some val), page-owned-by := old-pg.page-owned-by}*
H_{12}: *new-state = update-state-cache s (f-cache-add (fix-cache-synonym s.cache ma0) va new-pg)*

(update-aos-va-cont new-state va new-pg).cache =
f-cache-add (fix-cache-synonym s.cache ma0) va pg

Es fácil ver que como *update-aos-va-cont* solo modifica el campo *memory* entonces,

(update-aos-va-cont new-state va new-pg).cache = new-state.cache

Además, por hipótesis H_{12}, tenemos que,

new-state.cache = f-cache-add (fix-cache-synonym s s.cache ma0) va new-pg

Por lo tanto, por transitividad, la expresión a demostrar es equivalente a la expresión,

H_1: *valid-state s*
H_2: *Pre (s, write (va, val))*
H_3: *pg = {page-content := RW (Some val), page-owned-by := OS (s.active-os)}*
H_6: *Some ma0 = va-mapped-to-ma-cache s va*
H_7: *map-valid-index s.cache va = True*
H_8: *map-valid-index s.tlb va = True*
H_9: *map-valid-index s.memory ma0 = True*
H_{10}: *old-pg.page-owned-by = OS (s.active-os)*
H_{11}: *new-pg = {page-content := RW (Some val), page-owned-by := old-pg.page-owned-by}*
H_{12}: *new-state = update-state-cache s (f-cache-add (fix-cache-synonym s.cache ma0) va new-pg)*

---

f-cache-add (fix-cache-synonym s.cache ma0) va new-pg =
f-cache-add (fix-cache-synonym s.cache ma0) va pg

Por hipótesis H_{10}, es fácil ver que *new-pg = pg*. Por lo tanto, la expresión a demostrar es válida por reflexividad.

QED

5.4. Manejo de Errores

Si bien hemos visto que las acciones definidas en la sección 5.2 son correctas, estas no chequean que se estén cumpliendo sus precondiciones a la hora de ser ejecutadas. Esto significa que una acción podría ser ejecutada en un momento en el cual no están dadas las condiciones ideales para que esta sea ejecutada, efectivamente, de manera correcta. En esta sección veremos como redefinir las acciones[5] de manera tal que estas sí tengan en cuenta sus precondiciones y se analizará que tipos de errores podrían surgir al ejecutarlas. Cabe destacar que como el análisis

[5]Solo se mostrará con las acciones que fueron dadas en la sección 5.2. Todas las acciones redefinidas pueden ser vistas en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`.

en profundidad de todo lo relacionado con las ejecuciones inválidas no forma parte de los objetivos de este trabajo solo se sentarán sus bases. Esto facilitará la tarea de un futuro trabajo que sí haga dicho análisis.

5.4.1. Ejecución Inválida de una Acción

La ejecución inválida de una acción *act* en un estado válido *s* no debe modificar dicho estado. Esto es, representando la ejecución inválida con la relación $\widehat{\hookrightarrow}$,

$$\frac{\textit{valid-state(s)} \qquad \neg\ \textit{Pre (s, act)} \qquad \exists\ \textit{ec:ErrorCode, ErrorMsg s act ec}}{s\ \widehat{\hookrightarrow}_{act}\ s}$$

,donde *ec* representa a un código de error que se asocia a un posible fallo que pueda producirce a la hora de ejecutar una acción y *ErrorMsg* es un predicado que corrobora que *ec* sea el código de error asociado al fallo de la acción *act* en el estado *s*.

Códigos de Error

Teniendo en cuenta solo las acciones definidas en la sección 5.2, el conjunto de todos los códigos de error se define como,

$$\textit{ErrorCode} := \{\ \textit{invalid-vadd, wrong-page-type, no-access-va-os, no-access-va-hyp, os-non-waiting, os-non-running, pending-hcall}\ \}$$

Estos códigos de error significan lo siguiente:

- ***invalid-vadd:*** No existe una dirección de máquina asociada a la dirección virtual analizada.
- ***wrong-page-type:*** El tipo de que posee una página no es el tipo esperado.
- ***no-access-va-os:*** La dirección virtual analizada no es accesible por el guest OS activo (y debería serlo).
- ***no-access-va-hyp:*** La dirección virtual analizada no es accesible por el **VMM** (y debería serlo).
- ***os-non-waiting:*** El guest OS activo no está en espera (y debería estarlo).

- ***os-non-running:*** El guest OS activo no está corriendo (y debería estarlo).

- ***pending-hcall:*** Hay una *hypercall* esperando por ser atendida (y no debería estarlo).

Predicado *ErrorMsg*

Este predicado será representado con la siguiente tabla en la cual *act* representa a la acción a ejecutar y *ec* representa al código de error que corresponde al posible fallo ocurrido al ejecutar *act* en el estado *s*:

act	Fallo	*ec*
read (va)	*context.os-accessible[va] <> True*	*no-access-va-os*
	s.aos-activity <> running	*os-non-running*
	¬ ∃ (ma:madd), va-mapped-to-ma (s, va, ma)	*invalid-vadd*
	page-type (s.memory[ma]) <> RW	*wrong-page-type*
write (va, val)	*context.os-accessible[va] <> True*	*no-access-va-os*
	s.aos-activity <> running	*os-non-running*
	¬ ∃ (ma:madd), va-mapped-to-ma (s, va, ma)	*invalid-vadd*
	page-type (s.memory[ma]) <> RW	*wrong-page-type*
switch (osi)	*¬ ∃ (system:os), Value system = s.oss osi ∧ system.hcall = None*	*pending-hcall*
	s.aos-activity <> waiting	*os-non-waiting*

5.4.2. Acciones

Para redefinir una acción primero definiremos una función la cual chequee el cumplimiento de las precondiciones de la acción en cuestión. Dicha función devolverá *None* si la precondición no se cumple y *Some s'* si la precondición se cumple, siendo *s'* el estado que cumplió con la precondición[6]. Luego, combinaremos dicha función con la acción ya definida en la sección 5.2 para generar una nueva acción a la cual llamaremos ***"acción segura"***. Las acciones seguras tendrán un esquema similar al siguiente:

[6] *s'* = *s* si *s* cumple con la precondición

action-safe (s:state) : state :=
match *action-pre s* ***with***
| *None* ⇒ *s*
| *Some s'* ⇒ *action s*
end

, donde *action-pre* será la función que chequee si se cumple la precondición de la acción que se está definiendo y *action* será la definición de la acción dada en la sección 5.2.

Ahora, cuando la precondición falle no se ejecutará la acción, por lo que no se modifica el estado. Cabe destacar que en algunos casos las definiciones de las funciones que chequean las precondiciones no son directas:

- En la acción *new-hyper-action-safe* agregue el argumento *osi:os-ident* para poder incluir la precondición,

  ∀ (osi:os-ident), (∃ os, Value os = s.oss[osi] ⇒ ∃ p2m, Value p2m = s.hypervisor[osi])

- En las acciones *page-pin-untrusted* y *page-pin-trusted* (con y sin errores) se tuvo que agregar el argumento *ma:madd* el cual representa la dirección libre a utilizar (se hacen los chequeos correspondientes para asegurar que *ma* efectivamente está libre).

- En las acciones *page-unpin-trusted-safe* y *page-unpin-untrusted-safe* se tuvo que agregar los argumentos *pt-addr:madd* y *va:vadd* para poder incluir la precondición,

  no-va-mapped-to-ma osi ma s : Prop :=
  ∀ *pt-addr pt-page va mp,*
  Value pt-page = s.memory[pt-addr] ⇒
  pt-page.page-owned-by = OS osi ⇒
  pt-page.page-content = PT mp ⇒
  mp[va] <> Value ma.

Acción Read

La función que chequea la precondición de la acción *read* se define de la siguiente manera:

read-action-pre (s:state) (va:vadd) : option state :=
match *get-aos-ma s va* ***with***

```
    | None ⇒ None
    | Some ma ⇒
            match page-type (s.memory[ma]) with
              | Some RW ⇒
                      match s.aos-activity with
                          | waiting ⇒ None
                          | running ⇒ if context.os-accessible[va] then Some s else None
                      end
              | - ⇒ None
            end
  end
```

En la misma, la función *get-aos-ma* se encarga de chequear la existencia de una dirección de máquina *ma* asociada a *va*, la función *page-type* chequea que el tipo de la página asociada a *ma* sea lectura/escritura (*RW*), se chequea el guest OS activo este corriendo y se comprueba *va* sea accesible por dicho OS. Luego, esta acción se define como,

```
  read-action-safe (s:state) (va:vadd) : state :=
match read-action-pre s va with
    | None ⇒ s
    | Some s' ⇒ read-action s va
  end
```

Acción Write

La función que chequea la precondición de la acción *write* se define de la siguiente manera:

```
  write-action-pre (s:state) (va:vadd): option state :=
match get-aos-ma s va with
    | None ⇒ None
    | Some ma ⇒
            match page-type (s.memory[ma]) with
              | Some RW ⇒
                      match s.aos-activity with
                          | waiting ⇒ None
```

```
                    | running ⇒ if context.os-accessible[va] then Some s else None
                end
            | - ⇒ None
        end
end
```

En la misma, la función *get-aos-ma* se encarga de chequear la existencia de una dirección de máquina *ma* asociada a *va*, la función *page-type* chequea que el tipo de la página asociada a *ma* sea lectura/escritura (*RW*), se chequea que el guest OS activo este corriendo y se comprueba que *va* sea accesible por dicho OS. Luego, esta acción se define como,

```
write-action-safe (s:state) (va:vadd) (val:value): state :=
match write-action-pre s va with
    | None ⇒ s
    | Some s' ⇒ write-action s va val
end
```

5.4.3. Pruebas de Corrección

Los teoremas a demostrar son casi idénticos a los vistos en la sección 5.3. La única diferencia es que el nuevo estado se obtiene luego de aplicar la acción definida como acción segura.

Teorema act-safe-correcta: ∀ (s:state) (act:action),
valid-state s ⇒ Pre (s, act) ⇒ Post (s, act, f-act-safe s)

, donde *f-act-safe* es la implementación de la acción *act* con manejo de errores y *f-act-safe s* representa al estado resultante luego de ejecutar la acción *act*.

Además, las pruebas de corrección de las acciones seguras no distan mucho de las pruebas vistas para las acciones sin manejo de errores. Ahora, comenzaremos las pruebas haciendo análisis por casos en el resultado obtenido al aplicar la función que chequea si se cumple la precondición de la acción en cuestión. Si se cumple, entonces la demostración es exactamente igual a la vista en la sección 5.3.3. Si no se cumple, como entre nuestras hipótesis tenemos que la precondición sí se cumple, entonces utilizamos dicho hecho para armar una contradicción y demostrar este caso por el ***absurdo***.

Cabe destacar que el hecho de que las definiciones de las funciones que chequean las precondiciones no son directas hace que en algunos casos tengamos que agregar algunos hipótesis a la prueba de corrección para poder llevarlas a cabo:

- En los casos en los que las precondiciones tienen una implicación cuantificada universalmente, como en *new-hyper* y *page-unpin*, hubo que incluir como hipótesis los antecedentes de dichas implicaciones para poder concluir las pruebas. Por ejemplo, para probar la corrección de la acción *new-hyper-action-safe* hay que demostrar el teorema,

  Teorema *new-hyper-spec-holds-safe: forall (s: state) (osi:os-ident) (os':os) (va:vadd) (ma:madd),* valid-state ⇒ Pre (s, new-hyper va ma) ⇒
  Value os' = s.oss[osi] ⇒ Pos (s, new-hyper va ma, new-hyper-action-safe s va ma osi)

- En las demostraciones de corrección de las acciones *page-pin* hubo que incluir como hipótesis la existencia de una *ma:madd* libre. Por ejemplo, en la prueba de corrección de *page-pin-untrusted* hubo que demostrar el teorema,

  Teorema *page-pin-untrusted-spec-holds: forall (s: state) (osi:os-ident) (pa:padd) (type:ptype), valid-state s ⇒ Pre (s, page-pin-untrusted osi pa type) ⇒*
  exists (ma:madd) (pg:page),
  Value pg = s.memory[ma] ∧ pg.page-owned-by = No-Owner ∧ pg.page-content = Other ⇒
  Pos (s, page-pin-untrusted osi pa type, page-pin-untrusted-action s osi pa type ma)

5.5. Derivación de Código Ejecutable Certificado

Luego de haber realizado todas las pruebas de corrección estamos en condiciones de afirmar que todas las implementaciones realizadas en las secciones anteriores son correctas según su especificación. Sin embargo, el código utilizado al trabajar con COQ no es ejecutable. ¿Esto significa qué si queremos utilizar nuestro modelo debemos reescribirlo por completo un lenguaje de programación adecuado (i.e. que pueda compilarse)?

No necesariamente. Afortunadamente COQ nos brinda herramientas para poder extraer código ejecutable en un lenguaje funcional a partir de una especificación escrita en dicho asistente de pruebas. El lenguaje por defecto de extracción es **OCaml**, aunque también pueden ser elegidos los lenguajes **Scheme** y/o **Haskell**.

En esta sección se explica cómo se avanzó un paso más en dirección de contar con un prototipo ejecutable del modelo, al extraer código ejecutable a partir de la especificación del modelo realizada en este trabajo. El lenguaje elegido para la extracción de código ejecutable será **Haskell** [14].

5.5.1. Inferencia de Código

Si bien COQ infiere el código funcional del componente pedido, así como los elementos de los cuales depende para su definición, hay que prestar especial atención a la extracción de los tipos y valores genéricos (e.g. *vadd, madd*, etc) y a la extracción las funciones que comparan por igualdad dichos valores. Esto es así dado que los tipos genéricos serán extraídos como el tipo *unit* (i.e. Data () = ()). Por lo tanto, hay que asociar un tipo específico a aquellos tipos que se definieron de manera genérica para que, de esta forma, la ejecución del código sea lo más real posible. En este trabajo se realizaron las siguientes asignaciones:

- A todos los tipos de direcciones de memoria y a los identificadores de guest OSs se les asignó el tipo entero (i.e Int).
- Al tipo *value* (i.e. contenido de las páginas R/W) se le asignó el tipo carácter (i.e. Char).
- Las funciones de igualdad se asociaron a la función polimórfica == provista por el preludio de Haskell.
- El tipo booleano de Coq se asoció al tipo booleano de Haskell (i.e. Bool).
- Suponiendo que cada bloque en la Cache tiene tamaño 4 bits (y almacena una sola página) y que la Cache es de 64 KB, entonces a la constante *max-cache* se le asigna el valor (de tipo entero) 131072 (número máximo posibles de bloques).

  Suponiendo que cada bloque en la TLB tiene tamaño 4 bits y que la TLB es de 16 KB, entonces a la constante *max-tlb* se le asigna el valor (de tipo entero) 32768 (número máximo posibles de bloques). Cabe destacar que como en la especificación se utilizan naturales en lugar de enteros, se ha agregado la función intToNat :: Int −> Nat en los archivos extraídos la cual transforma enteros en naturales. De esta forma se logra que todas las funciones tengan los tipos correctos.

5.5.2. Resultado de la extracción

El resultado de la extracción son 2 archivos llamados ***virtualCert.hs*** y ***virtualCert_Safe.hs***. El primero comprende a las acciones sin manejo de errores y el segundo comprende a las acciones seguras (i.e. con manejo de errores). Ambos archivos incluyen la siguiente línea de código:

import qualified Prelude

La misma busca que no se extraiga código ambigüo respecto de las definiciones de tipos y funciones que provee el Preludio[14] de Haskell. Sin embargo, para extraer el código con los cambios mencionados en la sección 5.5.1 necesitamos poder utilizar algunos de los tipos del Preludio de Haskell, por lo que dicha línea de código es cambiada por la siguiente línea:

import Prelude hiding(length)

Eliminar la palabra qualified nos permite utilizar sin problemas las definiciones del Preludio de Haskell. Ahora bien, esto ocasiona un problema de ambigüedad con la función del Preludio *length* que cálcula la longitud de una lista. Para evitar dicho problema, agregamos a la línea anterior hiding(length). Esto hace que al importar el Preludio la función length no sea tenida en cuenta.

Cabe destacar que para simplificar la lectura de la extracción de las distintas acciones, todas estas también han sido extraídas de manera particular en archivos cuyo nombre coincide directamente con el nombre de la acción que ha sido extraída en los mismos. Por ejemplo,

read-action.hs
read-action-safe.hs

Además, en los archivos extraídos hubo que modificar la identación de las acciones *new-untrusted-action-safe* y *page-pin-untrusted-action-safe* para que las estas sean reconocidas por el compilador de Haskell correctamente.

Todos los archivos antes mencionados se encuentran en `http://www.fing.edu.uy/inco/grupos/gsi/proyectos/virtualcert.php`, en el archivo **Codigo Extraido.rar**.

Capítulo 6

Análisis de variantes sobre el modelo formalizado

6.1. Tipos de Cache

Este trabajo estuvo basado en la primeras versiones de Xen ARM las cuales usaban una Cache VIVT. Sin embargo, desde la versión 6 de ARM (ARMv6) dicha Cache fue cambiada por una Cache VIPT. De esta manera, redujeron considerablemente la cantidad de flushings realizados en las versiones previas dado que, por ejemplo, solo se realizarán flushing de la Cache al cambiar de un proceso a otro si el hardware subyacente no soporta el uso de Cache VIPT. Introducir dicho cambio en nuestra especificación conllevaría realizar algunas modificaciones en la misma:

Direcciones

Podría haber que cambiar la representación de las direcciones, tanto físicas como virtuales, para poder obtener a partir de estas su índice o su etiqueta. Algunas maneras de hacer lo anterior podrían ser,

1. *Record Madd: Set := madd {tag:Bits ; ind:Bits ; offset:Bits}*

2. *Record Ind : Set := index {f2b:Bits ; n6b:Bits}*
   Record Madd: Set := madd {tag:Bits ; ind:Ind ; offset:Bits}

3. *Definition Madd := list Bits*

4. Definiciones de alto nivel

En 1, la idea es dividir las direcciones en la cantidad de bits necesarios para formar la etiqueta (18 bits) de una dirección, cantidad de bits necesarios para formar el índice (8 bits) y el offset que representa los bits restantes (6 bits). Esta representación podría ocasionar problemas a la hora de implementar la traducción de direcciones virtuales dado que al traducir el vpn (número de página virtual) se obtiene el TAG (etiqueta) de la dirección física y los 2 primeros bits del campo IND (índice).

Una posible solución del problema antes mencionado podría ser pensar el campo IND dividido en sus 2 primeros bits, los cuales están involucrados en el proceso de traducción, y los 6 bits restantes como se muestra en 2.

En 3, en lugar de pensar las direcciones respecto de los campos necesarios estas son pensadas simplemente como una lista de bits. Luego, habría que definir funciones que tomen los bits necesarios de la dirección para obtener su campo TAG ó IND, entre otras.

En 4, se trata de mantener la especificación con el más alto nivel posible. En otras palabra, en lugar de modificar como se definen las direcciones de máquina se define *tag* e *ind* como funciones abstractas (e.g. $tag\_madd := madd \rightarrow TAG$).

Cache

Habría que cambiar la función de correspondencia de la Cache dado que la función de correspondencia asociativa no tiene en cuenta el campo índice. Además, en base a dicha modificación, podría llegarse a tener que cambiar su representación (estructura).

Si se elige una función de correspondencia directa, es posible mantener la Cache como un mapping, pero esto haría que el acceso a los elementos no sea en tiempo constante, hecho que es deseable en este tipo de correspondencia. Una opción a tener en cuenta podría ser cambiar la estructura de la Cache de un mapping a una random access list, lo que mejoraría los tiempos de acceso a sus elementos.

Si se elige una función de correspondencia asociativa por conjuntos, mantener la Cache como un mapping de madd a page dificultaría demasiado la manipulación de la misma. Podría ser más útil pensar la Cache como un record cuyas componentes sean todas mappings de madd a page y asociar a cada una de estas un valor de acceso (IND mod max_cache). Esto último haría que

haya que agregar el invariante de que la suma del tamaño de todas las componentes de dicho record no sea mayor al tamaño máximo de la Cache.

Cabe destacar que dichos cambios influirían directamente en la implementación de *f-cache-add* y de *f-cache-drop*.

TLB y Tablas de Paginación

Estas no sufrirían cambios significativos. Solamente habría que modificar un poco la manera en que se realizan las traducciones de las direcciones virtuales, al haberse dividido la vista de las direcciones en partes (como se mencionó anteriormente).

Acciones

Habría que cambiar un poco la implementación de algunas de las acciones que interactúan directamente con la Cache:

- *read-action* y *read-hyper-action*

  Estas acciones ahora para poder chequear si la página asociada a la dirección virtual a la que se quiere acceder está en la Cache, primero deben buscar la traducción del campo TAG y, de esta manera, poder chequear si la dirección ubicada en el índice IND de la Cache es realmente la que está asociada a la dirección virtual en cuestión.

- *write* y *write-hyper*

  Estas acciones ahora no necesitan chequear si hay sinónimos de direcciones en la Cache (y corregir dicho problema) dado que como las direcciones de máquinas tienen un único campo TAG [20], se utiliza dicho campo para saber a que dirección se está haciendo referencia.

- *switch* y *lswitch*

  Como se mencionó anteriormente, dado que las direcciones de máquinas tienen un único campo TAG estas acciones ahora no necesitan realizar un flushing de la Cache al producirse un cambio de contexto.

6.2. Política de Escritura de la Cache

Actualmente se está utilizando una política de escritura *Inmediata (Write-Trough)* en la Cache. Esto incentivó, por ejemplo, a que se defina un invariante de gran uso en el modelo que dice que toda página que se encuentra en la Cache también se encuentra en la memoria. Sin embargo, se podría haber optado por utilizar una política de escritura *Demorada (Write-Back)*. Dicha elección hubiese significado tener que analizar las modificaciones introducidas al modelo en este trabajo de diferente manera. Por ejemplo, el invariante antes mencionado dejaría de ser válido dado que con la política de escritura *Demorada* una página en la Cache puede ser actualizada sin que lo sea la correspondiente página en la memoria. Esto significa, que no necesariamente el contenido de una página en la Cache sería igual en todo momento a su contenido en la memoria. A continuación serán mencionados algunos de los cambios que habría que tener en cuenta si se quisiera cambiar a *Demorada* la política de escritura actual.

Cache

Habría que cambiar la representación de la Cache para tener en cuenta el *dirty bit* (ver sección 2.2.2 - Escritura Demorada). Una forma sencilla de hacer esto podría ser pensando a la Cache de la siguiente manera:

$$dirtyBit := \{0,\ 1\}$$
$$cache\text{-}struct := vadd \rightarrow \{pagina{:}page,\ db{:}dirtyBit\}$$
$$cache : cache\text{-}struct$$

, donde *pg* será la página asociada a la dirección virtual recibida como argumento y *db* representará el dirty bit correspondiente a *pg*.

Además, habría que modificar la operación *f-cache-add* para que al agregar la página *pg* a la Cache en *va*, se agregue el record,

$$\{pagina := pg,\ db := 0\}$$

, si *va* no estaba en la Cache o, en caso contrario, que se agregue el record,

$$\{pagina := pg,\ db := 1\}$$

Acciones

En este caso la mayoría de las acciones no necesitarían cambios dado que para interactuar con la Cache estas utilizan las operaciones provistas por la antes mencionada (e.g. *f-cache-add*). Las únicas acciones que necesitarían especial atención son *write* y *write-hyper* por los siguientes motivos:

- Al escribir en una página que ya se encuentre en la Cache solamente hay que actualizar la Cache. La memoria en estos casos no se modifica en lo absoluto.
- Al intentar escribirse en una página que no está en la Cache se producirá un **Cache miss**. Por lo tanto, todas las páginas cuyo valor de dirty bit asociado sea 1 deben ser actualizadas en la memoria.
- Al escribir una página que no este en la Cache dicha página debe agregarse en la anterior con valor de dirty bit 1. Pero como *f-cache-add* la agregaría con valor 0 habría que, o bien utilizar una función que dada una dirección virtual ponga el dirty bit de su página asociada en la Cache en 1, o bien agregar dos veces la misma entrada (la primera vez se agrega a la Cache $\{pagina := pg,\ db := 0\}$ y la segunda vez se cambia el valor de dirty bit a 1, i.e. $\{pagina := pg,\ db := 1\}$) en la Cache.

Pruebas de Corrección

Actualmente, gracias al invariante que dice que toda página que se encuentra en la Cache también se encuentra en la memoria, podemos demostrar por el absurdo de manera trivial todos los casos en las pruebas de corrección cuyas hipótesis violen dicho invariante. En cambio, si se utilizará una política de escritura *Demorada* dichos casos deberían ser analizados en profundidad dado que el invariante antes mencionado carecería de validez.

Invariantes del Modelo

Si bien el invariante que dice que toda página que se encuentra en la Cache también se encuentra en la memoria cuando se utiliza una política de escritura *Demorada* carece de validez, se podría definir un invariante similar que diga que si el dirty bit de una página que se encuentra en la Cache vale 0 entonces dicha página también se encuentra en la memoria.

Capítulo 7

Conclusiones y Trabajo a Futuro

Se presentó un resumen del reporte técnico realizado al estudiar el estado del arte del uso de modelos de memoria en plataformas de virtualización.

Se presentó el modelo idealizado de un **VMM** diseñado en el proyecto VirtualCert, el cual se basa en una máquina de estados, y que pone énfasis en garantizar un acceso seguro y aislado a los datos en memoria, por parte de los guest OSs. Dicho modelo fue formalizado usando COQ, por lo que las propiedades que deba cumplir el modelo pueden ser expresadas como predicados lógicos y ser demostrados matemáticamente mediante el uso de este asistente de pruebas.

Se extendió el modelo idealizado de plataforma de virtualización actual del proyecto VirtualCert agregando el uso de Cache y TLB al estado. Para esto, se modeló la estructura de la Cache y la TLB, fueron implementadas operaciones para agregar y quitar elementos de estas y fue probado que dichas operaciones son correctas según su especificación. Además, se analizó el hecho de que dichas operaciones no pueden ser independientes del estado.

Al haberse modificado el estado, fueron reimplementadas las distintas acciones de la plataforma para que estas tengan en cuenta que se está haciendo uso de Cache y TLB (sin y con manejo de errores) y se demostró formalmente que eran correctas según su especificación. Además, en base a estas nuevas implementaciones, se extrajo un prototipo ejecutable del **VMM** en el lenguaje Haskell gracias a las herramientas de COQ.

Por último, se analizó de que manera influiría en el modelo la introducción de ciertos cambios en su especificación y se mencionaron algunos proyectos relacionados con lo hecho en este trabajo.

Como trabajos futuros, se abren diferentes caminos que pueden aportar mucho valor a lo

alcanzado hasta el momento en el proyecto VirtualCert:

- Adaptar la especificación dada en este trabajo para que el tipo de Cache utilizado sea opcional. Es decir, poder elegir entre una Cache VIVT y una Cache VIPT. Esto apunta a que cuando se virtualice un sistema operativo que no soporta el uso de una Cache VIPT la plataforma trabaje como lo hace en este trabajo (con una Cache VIVT). En cambio, si el sistema operativo a virtualizar si soporta el uso de Cache VIPT entonces que la plataforma trabaje utilizando dicho tipo de Cache.
- Estudiar en profundidad todo lo relacionado con las (posibles) ejecuciones inválidas que pueden darse en el sistema.
- Estudiar en profundidad que repercuciones podría tener cambiar la política de escritura de la Cache.
- Agregar al modelo nociones de entrada y salida.
- Abordar el soporte para ejecución concurrente en ambientes multi-core.

Bibliografía

[1] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

[2] Eyad Alkassar, Mark A. Hillebrand, Wolfgang J. Paul, and Elena Petrova. Automated verification of a small hypervisor. In *Proceedings of the Third international conference on Verified software: theories, tools, experiments*, VSTTE'10, pages 40–54, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven H, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP (2003*, pages 164–177.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Springer.

[5] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *ACM Transactions on Computer Systems*, pages 143–156, 1997.

[6] S. K. Gupta D. Engler and F. Kaashoek. Avm: Application-level virtual memory. In *In Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, May 1995.

[7] Markus Dahlweid, Michal Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. Vcc: Contract-based modular verification of concurrent c.

[8] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2:153–189, 1970.

[9] U. Meyer et al. Algorithms for memory hierarchies. Technical report, LNCS 2625, 2003.

[10] J. D. Campo C. Luna G. Barthe, G. Betarte. Formally verifying isolation and availability in an idealized model of virtualization. *FM 2011: 17th International Symposium on Formal Methods, Ireland*, 2011.

[11] J. M. Chimento C. Luna G. Betarte, J. D. Campo. Modelos de memoria en entornos de virtualización. Technical report, 12-02, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2012.

[12] Robert Goldberg and Robert Hassinger. The double paging anomaly.

[13] Robert Goldberg and Gerald Popek. Formal requirements for virtualizable third generations architectures. In *Communications of the ACM, 17(7)*, July, 1974.

[14] Simon Peyton Jones. *Haskell 98 language and libraries: the Revised Report.* Cambridge University Press.

[15] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pages 207–220. ACM, 2009.

[16] R. A. Nelson L. A. Belady and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. In *Communications of the ACM, 12(6)*, June 1969.

[17] Cuong Hoang H. Le, July 2009. Protecting Xen hypercalls. Master of Science Thesis, University of British Columbia (Vancouver).

[18] Dirk Leinenbach and Thomas Santen. Verifying the microsoft hyper-v hypervisor with vcc. In *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, pages 806–809, Berlin, Heidelberg, 2009. Springer-Verlag.

[19] Gustavo Betarte Nora Szasz, Cristina Cornes and Alvaro Tasistro. Specification of a smart card operating system.

[20] David Patterson and John Hennessy. *Computer Organization and Desing, Third Edition.* Morgan Kauffman.

[21] Robert Rose. Survey of system virtualization techniques. Technical report, 2004.

[22] Sangwon Seo. Research on system virtualization using xen hypervisor for arm based secure mobile phones. Technical report, Seminar "Security in Telecommunications", Berlin University of Technology, Korea Advanced Institute of Science and Technology, January 2010.

[23] Silberschatz and Galvin. *Operating Systems Concepts, Fourth Edition.* Addison Wesley.

[24] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14:473–530, 1982.

[25] William Stallings. *Operating Systems, Second Edition.* Prentice Hall.

[26] Dominic Sweetman. *See Mips Run, First Edition.* Morgan Kauffman.

[27] Andrew Tanenbaum. *Modern Operating System, Second Edition.* Prentice Hall, 2002.

[28] The Coq Development Team, 2009. The Coq Proof Assistant Reference Manual.

[29] Ruud van der Pas. Memory hierarchy in cache-based systems, 2002.

[30] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[31] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *In Proceedings of the USENIX Annual Technical Conference*, 2002.