Hermes: DSM por software con granularidad fina

Horacio Andrés Lagar Cavilla Rafael Benjamín García

LISiDi – Laboratorio de Investigación en Sistemas Distribuidos
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur, Av. Alem 1253, (8000) Bahía Blanca, Argentina
Tel ++54 291 4595135 - Fax: ++54 291 4595136
{alc,rbg}@cs.uns.edu.ar

Resumen

Los sistemas de DSM por software sobre clusters de workstations constituyen una alternativa interesante para el procesamiento paralelo, gracias a su gran potencial para la escalabilidad y excelente relación costo/performance. Al encarar el diseño de un sistema de este tipo, los aspectos fundamentales a considerar son: que sea simple y eficiente, que pueda utilizarse sobre una plataforma estándar sin requerimientos onerosos de hardware, que minimice el efecto negativo de *false sharing* proveniente de la gruesa granularidad de consistencia que acarrea el uso del mecanismo de protección de memoria virtual, y que evite las latencias prohibitivas asociadas al tráfico de mensajes sobre protocolos de red estándares durante las etapas de sincronización. En este trabajo presentamos **Hermes**, un sistema de DSM por software que resuelve la problemática citada con un enfoque simple y totalmente novedoso, al evitar el costo de mantener un orden parcial de las referencias –con ciertos modelos de consistencia relajados–. Hermes provee además control de granularidad fina de complejidad y sobrecarga mínima, que potencia su escalabilidad y brinda a su vez una alta flexibilidad para utilizar el modelo de consistencia que resulte más apropiado. En lo que respecta a la comunicación, proponemos recurrir al uso de interfaces de red mapeadas a memoria virtual.

Palabras clave: Procesamiento paralelo, DSM escalable, granularidad fina, consistencia relajada, cache virtual.

1. Introducción

A principios de la década pasada surge dentro del campo del procesamiento en paralelo una alternativa a los dos paradigmas tradicionales de multicomputadoras con pasaje de mensajes y multiprocesadores con memoria compartida. Los sistemas de DSM (*Distributed Shared Memory* – Memoria Compartida Distribuida) intentan enmascarar la necesidad de enviar explícitamente mensajes entre los nodos componentes de un sistema distribuido, al proveer la ilusión de un espacio global de memoria compartida por todos los procesadores. La idea subyacente implica interceptar los accesos a datos residentes en memorias remotas, y de forma transparente traducirlos en mensajes sobre la red de interconección. El nuevo paradigma propone entonces un compromiso entre las dos alternativas tradicionales, apoyándose en la facilidad de implementación del pasaje de mensajes e intentando proveer la simplicidad de programación que brinda la memoria compartida. La evolución de los sistemas de DSM se desarrolló en dos ramas claramente diferenciadas:

- Sistemas de DSM por hardware con un control de granularidad fina ejercido en el cache, y con una red de interconección propietaria de muy baja latencia.

- Sistemas de DVSM, o Memoria Virtual Compartida Distribuida, compuestos por un cluster de workstations en las que se utiliza el hardware de protección de memoria virtual para implementar un control de granularidad mucho más gruesa, y se utilizan substratos y protocolos de comunicación estándares para la interconección de los nodos.

Dentro de la primera clasificación se diferencian dos paradigmas básicos: las arquitecturas COMA (*Cache Only Memory Architecture* - Arquitectura de Memoria Sólo Cache) y CC-NUMA (*Cache Coherent Non Uniform Memory Access* - Acceso a Memoria No Uniforme con Caches Coherentes). En las primeras se utilizan unos pseudo-caches llamados *memorias de atracción* que alojan bloques de cache compartidos utilizados actualmente por el procesador local. Al referenciar una variable no presente en la *memoria de atracción* local, el bloque de cache correspondiente migrará hacia la misma y permanecerá allí, minimizando la latencia de futuras referencias. Esta migración automática del *working set* hacia el procesador, aunada al principio de localidad, son los factores en los que se apoya esta propuesta para proveer una alta performance. Además, dado el carácter migratorio de los bloques no contamos en este caso con una noción estática de *pertenencia* de los bloques a un nodo del sistema. Sin embargo, ante la escritura a un bloque compartido se deben invalidar todas las copias del mismo –como en un SMP basado en bus–, por lo que futuras referencias a ese bloque generarán nuevos *misses* y transferencias. El primer prototipo de esta clase de arquitecturas fue el DDM [9]. En un sistema CC-NUMA como DASH [16], se utiliza el cache tradicional del procesador para alojar los bloques compartidos. El controlador de cache es el encargado de mantener la coherencia de los contenidos del mismo, en general ayudado por un directorio que favorece la localización rápida de los bloques que son *owned* –propiedad– del nodo local.

Un problema inherente a una configuración multiprocesador de memoria compartida es el de la consistencia de memoria. Si bien la consistencia secuencial SC [15] se presenta como el modelo más intuitivo, en cuanto generaliza la idea de coherencia que se aplica al cómputo secuencial, su adopción implica una penalización sustancial en complejidad y/o performance. Para posibilitar optimizaciones, tanto a nivel de arquitectura como de compilador, resulta útil relajar el requerimiento de visualizar todos los accesos a memoria como secuencialmente consistentes, lo cual con las prácticas de programación más usuales –libres de condiciones de carrera– no agrega mayor complejidad a la tarea del programador. Para los sistemas de DSM del segundo tipo, también llamados sistemas de DSM por software, cobra mayor importancia contar con un modelo de consistencia relajado para disminuir la necesidad de comunicación y actualizaciones entre los nodos del sistema.

En un sistema de DVSM tradicional como TreadMarks [12], se intercepta el primer acceso a una página de memoria virtual compartida a través del mecanismo de protección de la MMU, para en ese momento traer el contenido actualizado de la misma. El hecho de compartir información con una granularidad del tamaño de la página virtual trae a colación dos efectos indeseables, no presentes en las propuestas de hardware mencionadas. Por un lado está el *false sharing*, o compartimiento no deseado de variables cuya actualización remota puede desencadenar invalidaciones innecesarias y prohibitivas en términos de rendimiento. Por otro lado tenemos la fragmentación interna de una página, que se traduce en la transmisión de gran cantidad de información innecesaria en los momentos de actualización. Si a esto le sumamos que el sistema se ejecutará en general sobre workstations estándares y con una altísima latencia promedio en la transmisión de mensajes, nos topamos con un panorama poco optimista en cuanto a las posibilidades de esta propuesta.

Sin embargo, los sistemas de DSM por software cuentan a su favor con varios factores que convierten su investigación y desarrollo en una tarea provechosa. La utilización de modelos de consistencia relajados, como ya dijimos, permite ocultar la latencia incurrida en las comunicaciones al posponer y agrupar las mismas hasta el último momento necesario [13]. Además contamos con un potencial para la escalabilidad mucho más alto que en implementaciones de hardware, y la flexibilidad inherente a la concepción del sistema como código de software. Esta flexibilidad se manifiesta de varias formas, como la posibilidad de permitir múltiples escritores a una misma

página de memoria compartida [12], la utilización de una granularidad variable para la transmisión y consistencia de datos [1, 18], la expansión del sistema para soportar plataformas heterogéneas [4], la aplicación de varias optimizaciones sólo realizables en un modelo de software [18], e inclusive la utilización de varios modelos de consistencia en simultáneo [2]. Sin embargo, quizás el aliciente más importante para el desarrollo de sistemas de DSM por software sea que los mismos se pueden implementar y ejecutar sobre un cluster de workstations estándares. Las mejoras progresivas en performance de los procesadores y redes de comunicación comerciales han permitido considerar dicha configuración como una alternativa válida y accesible para el procesamiento paralelo, en comparación con multiprocesadores con una similar cantidad de procesadores, mejor performance, pero una diferencia de costo sustancial. Las características de portabilidad, ubicuidad, y sobre todo bajo costo que sistemas como TreadMarks revelan, hicieron de los sistemas de DSM por software una alternativa de importancia creciente ante la poca esperanza de vida en términos comerciales de complejos y costosos diseños de hardware, como el KSR1 [6].

En este trabajo proponemos un sistema de DSM por software que intenta ofrecer todos los aspectos positivos señalados en distintas propuestas del área. A diferencia de los sistemas de DVSM tradicionales, nosotros contamos con un control de granularidad fina sobre los datos compartidos, basado en la atención por software de los *misses* de cache en una jerarquía de caches con índice y tag virtuales. No es la primera vez que se diseña un sistema de software con granularidad fina, y los mismos han probado ofrecer una performance comparable con ambiciosos sistemas de DSM en hardware [18, 19]. A diferencia de las propuestas anteriormente citadas, en **Hermes** el control de granularidad se incorpora de forma natural a la dinámica del sistema, integrándose al momento de la traslación de direcciones virtuales a físicas, sin incurrir en ninguna sobrecarga adicional ni obstaculizar la escalabilidad, y proveyendo un método simple para implementar cualquier modelo de consistencia que consideremos apropiado. De esta forma hemos obtenido lo mejor de ambos mundos: una implementación flexible, altamente escalable y con el potencial para múltiples mejoras y aplicaciones, pero efectuando de una forma sencilla y barata un control de granularidad fina típico de sistemas de hardware. Este control de granularidad fina efectuado en el cache le confiere a Hermes un comportamiento similar al de una arquitectura flat-COMA [5].

Consideramos utilizar en este trabajo el modelo de consistencia *Scope Consistency*, que nos brinda una interfaz de programación ya conocida y comparable a la del modelo *Release Consistency* [7], pero con una optimización de la performance y necesidad de comunicación cuasi óptima, similar a la del modelo *Entry Consistency* [2]. Presentamos una implementación para este modelo de consistencia sencilla y carente de estructuras de datos globales, que además prescinde de la problemática usual de mantener una relación de orden para los accesos de memoria compartida a través de estampillas de tiempo. Esta innovación significativa que proponemos resulta en una minimización sustancial de la necesidad de comunicación y procesamiento en los puntos de sincronización, trazando una clara diferencia con respecto a otros sistemas de DSM por software.

Por último resaltamos el hecho que un soporte de hardware reducido, como sería una interfaz de red mapeada a memoria virtual o el uso de un procesador dedicado, disminuiría sensiblemente el costo de envío de mensajes por la red de interconección. Con esta modificación prevemos una mejora cuantitativa de performance, lo cual tornaría a esta propuesta en una alternativa no solo accesible sino además eficiente para el procesamiento paralelo.

2. Hermes

El funcionamiento de Hermes se basa en la utilización de una jerarquía de caches de índice y tag virtuales (caches virtuales o V/V de aquí en adelante). Un cache V/V recibe referencias a direcciones virtuales directamente desde el procesador, sin mediar una traslación previa de dicha dirección virtual a su equivalente física. Dentro del cache se indexa con una porción de la dirección

virtual, mientras que la otra se usa para la comparación de tags. En caso de no encontrarse el bloque deseado en el cache, recién en ese momento será necesaria la traslación de la referencia virtual fallida a la dirección física correspondiente, para ubicar el dato dentro de la memoria principal. Si contamos además con una memoria cache de gran tamaño, el *hit ratio* se acerca al 100 % ideal, por lo que la necesidad de realizar traslaciones se ve sensiblemente disminuida. En [11] se prueba mediante simulaciones que si esa traslación se realiza mediante una rutina de software se puede obtener una performance comparable –inclusive ligeramente mejor en ciertos casos– a la de las plataformas con un esquema de traslación de direcciones tradicional. La utilización de un manejador o *handler* de *misses* por software nos brinda además flexibilidad absoluta en cuanto a las tareas a ejecutar para atender dicha falla.

Podemos integrar así de manera simple y efectiva un control de granularidad fina para nuestro sistema de DSM, aprovechando la funcionalidad del *handler* que se activará cuando es necesario atender un faltante de bloque de cache en el último nivel de la jerarquía. Dado que manejaremos la consistencia y transmisión de información a nivel de bloques de cache, como en una implementación de hardware, reducimos considerablemente el problema de *false sharing* del que adolecen la mayoría de los sistemas de DSM por software que manejan una granularidad del tamaño de la página de memoria virtual, y no necesitamos contar con un protocolo de múltiples escritores análogo al usado en TreadMarks. Además, el manejo por software de los *misses* permite implementar con facilidad cualquier tipo de organización de memoria virtual que deseemos, y cualquier tipo de modelo de consistencia que consideremos apropiado. Por último, podemos implementar de forma sencilla un *prefetching* implícito de la información compartida en el momento de resolución de un *miss* de cache, trayendo junto al bloque requerido múltiples bloques adicionales que presumiblemente serán utilizados a continuación –por ejemplo si la estructura de datos referenciada ocupa varios bloques–.

La arquitectura que proponemos para Hermes cuenta entonces con una jerarquía de caches virtuales que totalizan en el último nivel un tamaño en el orden de 4 a 16 MB. Ante la ocurrencia de un *miss* de cache se generará una excepción que pasará el control al *handler* adecuado[1]. Éste debe discernir si el *miss* se debe a un bloque alojado localmente o a un bloque de memoria compartida perteneciente a un nodo remoto. Para el primer caso, el *handler* ejecuta el algoritmo de recorrido de la tabla de páginas (*page table walking*) hasta obtener el PTE con la traslación correspondiente. Para el segundo caso se envía un mensaje al nodo correspondiente solicitando una copia del bloque, y de varios bloques adicionales si utilizamos *prefetching*. Mediante la adición al set de instrucciones de la instrucción especial de dos operandos *Mapped Load* (carga mapeada, ver figura 1), se puede ubicar el bloque obtenido en el cache virtual. Uno de los operandos (`Rv`) especificará la dirección virtual y bits adicionales del bloque, y el otro (`Rf`) indicará la dirección física desde donde se realizará la carga, extraída del PTE correspondiente para un bloque local, o del *buffer* de la interfaz de red para bloques pertenecientes a otro nodo. En la figura 2 se visualiza el funcionamiento de nuestro mecanismo de resolución de *misses* de cache.

```
Mapped Load (Rf,Rv) Rf -> Dirección física
                    Rv -> Dirección virtual y bits adicionales
```

Figura 1: Sintaxis de la instrucción *Mapped Load*

Nuestra intención es utilizar un espacio de direcciones virtuales global para todos los nodos del sistema, donde a cada nodo se le asigna propiedad sobre una porción equitativa de ese espacio de direcciones. Esta propuesta cobra mayor relevancia con el advenimiento de microprocesadores de 64 bits, que proveen un vasto espacio de direccionado. El *handler* de fallos podrá distinguir inmediatamente entonces entre referencias a datos locales o datos remotos, ya que los $log_2 N$ bits superiores de la dirección indicarán el nodo *home* para el bloque faltante (con N procesadores).

[1]Esta excepción puede ser la excepción usual de *TLB miss*, aunque no debe dispararse para el caso de un *miss* en el TLB con *hit* en el cache.

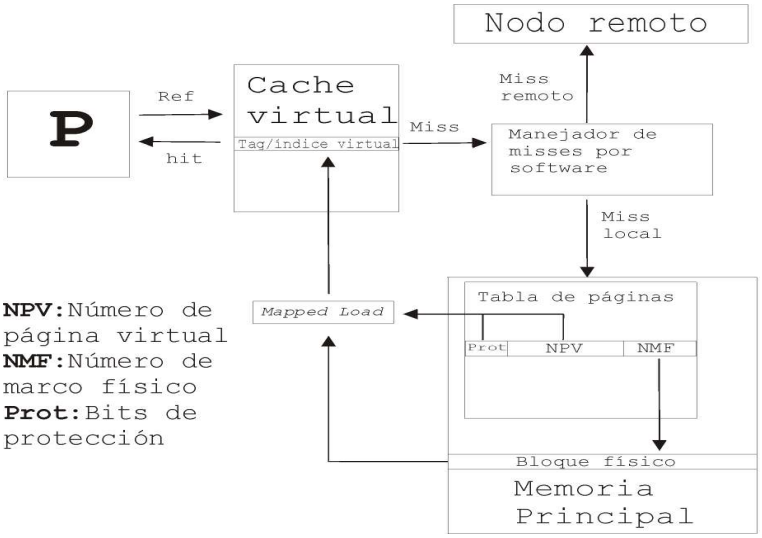

Figura 2: Esquema de resolución de *misses* de cache.

Nuestro cache utilizará una política de escritura *write-back*, donde sólo los bloques pertenecientes al espacio de direcciones local son escritos en memoria principal si al ser reemplazados se encuentra que fueron modificados (*dirty*). Los bloques pertenecientes a otros nodos y en estado *dirty* al ser reemplazados son, en principio, enviados a su nodo *home*. Éste manejo de los bloques de cache hace necesario contar con un modelo de consistencia de memoria relajado, y con anotaciones explícitas para los momentos de sincronización. De esta forma, Hermes adquiere un comportamiento similar al de una arquitectura flat-COMA, donde los bloques de memoria compartida se replican en el cache –no la *attraction memory*– del procesador donde son requeridos, y además tiene asignados un nodo *home* para facilitar su localización. La elevada latencia inherente a un *miss* relacionado con un bloque remoto hace perentorio disponer de un espacio de almacenamiento generoso para estos items, acentuando la necesidad de contar con un cache local de gran tamaño.

La utilización de un esquema de segmentado como el presente en el PowerPC [17] permite manejar inocuamente varios de los problemas que afectan tradicionalmente a las arquitecturas con caches virtuales. En este sistema, la parte superior de la dirección generada por un proceso se reemplaza por uno de los identificadores de segmentos asociados a dicho proceso. Se extiende así el ancho de la dirección virtual, y esta ya no apunta a un espacio de direcciones confinado al proceso, sino a un espacio de direcciones global común (2^{52} bytes para el PowerPC). Si dos procesos comparten un segmento, comparten transparentemente todo un rango de direcciones virtuales globales y las variables allí contenidas, existiendo además una protección implícita de accesos a ese segmento –que se traduce en la posesión del identificador sólo por procesos autorizados–, y una mayor flexibilidad en las posibilidades de mapeo de regiones de memoria. En la figura 3 vemos como se comparte información en una organización de memoria virtual segmentada. Al contar con un espacio de direcciones virtuales globales –que también se puede lograr con SOs de 64 bits con un único espacio de direcciones–, evitamos algunos problemas clásicos de los caches virtuales relacionados con la multiplicidad de mapeos entre direcciones virtuales y físicas. Por ejemplo, no vamos a alojar en el cache dos bloques con tags virtuales distintos pero mapeados a la misma región de memoria física –también llamados *sinónimos*–, no serán necesarios cambios de mapeo para compartir información –y evitaremos las invalidaciones de cache consecuentes–, y no habrá superposición entre los espacios de direcciones de distintos procesos –evitando también invalidaciones durante cambios de contexto–.

La traslación de direcciones por software sólo durante la atención de *misses* de cache permite considerar la eliminación del TLB en favor de un mayor tamaño de memoria cache, como se propone en el diseño original de [11]. Sin embargo, para acceder a la información de estado de una página de

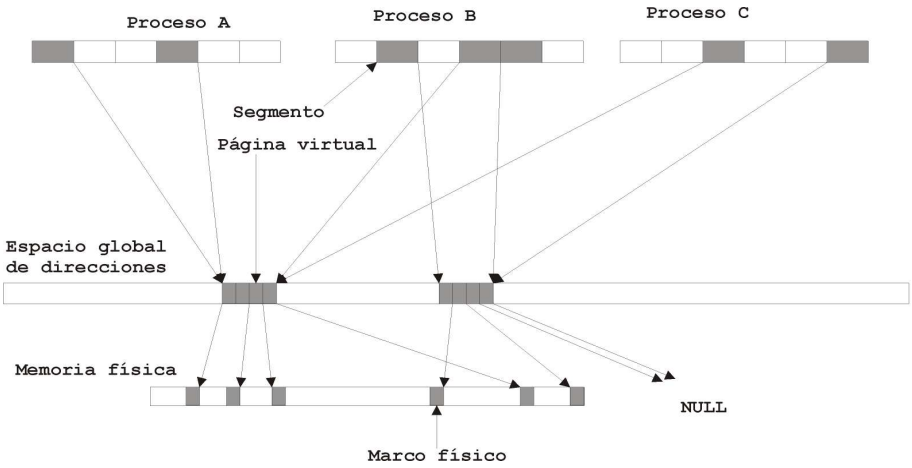

Figura 3: Compartimiento de información en una organización de memoria virtual segmentada.

memoria virtual –como los bits de protección–, deberemos replicar la misma para todos los bloques de cache que componen la página, y al ocurrir una modificación aplicarla en cada bloque presente en el cache. Dentro de la información que será necesario replicar debe incluirse además la traslación virtual a física para la página, necesaria al momento de propagar las escrituras en el cache a memoria principal. Asumiendo con una política *write back* que la mitad de los bloques reemplazados estarán *dirty*, no sería razonable ejecutar nuevamente la parte del *handler* que recorre la tabla de páginas para estos casos, porque el impacto en la performance global del sistema sería crítico. Podemos llegar al extremo de replicar el PTE completo dentro de cada tag del cache virtual, y posibilitar entonces la supresión total del TLB al cumplir su función la memoria asociativa de tags del cache virtual. Si bien creemos que en un ambiente DSM como el que aquí proponemos la frecuencia relativamente baja de cambios en la información de estado de una página disminuirá el costo incurrido por esa operación, no podemos olvidar que la utilización de arquitecturas estándar y la supresión de hardware adicional es una de las principales razones por las cuales se utilizan sistemas de DSM por software. Una solución más natural para este problema consiste en la utilización racional del TLB –apoyado en las capacidades de manejo por software del mismo provistas por la mayoría de las arquitecturas comerciales– para conservar las traslaciones de páginas cacheadas con permiso de escritura.

Para completar la funcionalidad de Hermes, requerimos ciertas capacidades de control del cache presentes también en varias arquitecturas comerciales. Es necesario efectuar *locking* de bloques de cache para evitar que las instrucciones del *handler* sean expulsadas durante su ejecución, generando un *miss* anidado e irresoluble. Por otro lado, la atención de un *miss* provocado por el *handler* implica soportar referencias físicas para el mismo –y también para atender otras operaciones cruciales–. Las referencias físicas obligan a identificar en el cache virtual bloques con tags físicos. Para estos bloques el sistema deberá mantener coherencia por hardware entre el cache y memoria principal. Caso contrario, durante una escritura asincrónica en memoria principal –una operación de E/S por ejemplo– se deben invalidar los bloques afectados y bloquear futuros accesos hasta el fin de la transferencia.

Hemos presentado en este capítulo un sistema que provee de forma automática un control de granularidad fina sobre la información, análogo al de implementaciones de DSM por hardware mucho más ambiciosas, y que además deja el juego abierto para desarrollar en una capa superior el o los modelos de consistencia que creamos apropiados. Esto se logra con una plataforma uniprocesador de performance comparable a aquellas con traslación de direcciones por hardware, y sin recurrir a ningún tipo de sobrecarga adicional, rutinas de software –más allá del *handler* de *misses*–, o estructuras de datos extras, características omnipresentes en los sistemas de DSM por software tradicionales. La única restricción que imponemos es que el modelo de consistencia deberá ser

relajado y con anotaciones explícitas de los puntos de sincronización. En el siguiente capítulo desarrollaremos una versión de Hermes que utilizará el modelo Scope Consistency.

3. Modelo de consistencia *Scope Consistency*

El modelo de consistencia Scope Consistency [10] que implementaremos sobre Hermes se basa en el uso de *contextos de consistencia*, entidades que encapsularán un conjunto de variables cuyo acceso debe ser sincronizado. Mediante el uso de operaciones *acquire* y *release* sobre un *lock* asociado, arbitramos el acceso a las variables contenidas en un contexto de consistencia, garantizando exclusión mutua al momento de efectuar las escrituras. Llamamos *sesión* de un contexto al fragmento de la ejecución del programa donde se realiza el *acquire* del *lock* correspondiente, se accede a las variables asociadas, y finalmente se libera el contexto mediante un *release*. La diferencia fundamental entre consistencia Scope y Release es que las variables son agregadas dinámicamente a un contexto a medida que se las accede dentro de una sesión, y que sólo las variables pertenecientes al contexto requerirán sincronización en el momento de un *acquire*.

Si bien esto puede obstaculizar el paso de programas escritos para consistencia Release a consistencia Scope debido a asunciones que ya no son válidas (referirse a los autores [10]), el modelo ofrece una performance similar al de la consistencia Entry, focalizando el gasto de comunicación y procesamiento durante etapas de sincronización en las variables que exclusivamente lo necesitan. Aparte de las primitivas *acquire* y *release* proveemos la tradicional primitiva *barrier* y la primitiva *create_scope*, una herramienta para crear dinámicamente nuevos contextos.

Previo al diseño de un protocolo para implementar el modelo de consistencia Scope, consideramos necesario formalizar ciertas situaciones indeseables y no previstas específicamente en el modelo. El objetivo fue aislar prácticas de programación erróneas que desencadenen condiciones de carrera en la ejecución de un programa en paralelo, y cuyo tratamiento implique además una sobrecarga adicional en la ejecución del protocolo e innecesaria para el caso usual de programas "correctos". Un programa escrito para el modelo de consistencia Scope Consistency que responda a las reglas a continuación enunciadas no sufrirá de dichas inconsistencias, y caerá dentro de la clasificación de los *properly labelled programs* [7].

Regla 1. Toda variable incluida en el ámbito de sesiones de uno o más contextos de consistencia no deberá ser accedida fuera de dichos contextos.

El objetivo de esta regla es evitar el acceso no arbitrado a variables para las cuales se intenta sincronizar su utilización. En particular, la escritura de una variable incluida en un contexto de consistencia fuera de la sección crítica generará una condición de carrera.

Regla 2. Toda variable que sufre modificaciones en el ámbito de una sesión de un contexto de consistencia no deberá ser modificada en el ámbito de una sesión de otro contexto de consistencia.

En este caso deseamos evitar la utilización de contextos anidados, y la inclusión de variables en más de un contexto de consistencia. Como vemos en la tabla 1, al permitir la escritura "legal" de una variable en múltiples ámbitos se genera también una condición de carrera.

Enunciaremos además una tercera regla que es más bien una expresión de deseo:

Regla 3. Toda variable que no este contenida dentro de un contexto de consistencia deberá ser una variable de sólo-lectura.

Para el modelo de consistencia Scope, Hermes no hace –ni debe hacer– ningún tipo de previsión

P0	P1	P2
Acquire(L0)		
Acquire(L1)		
X2=2		
Release(L1)		
Release(L0)		
	Acquire(L1)	Acquire(L0)
	X2=4	X2=X2+1
	Release(L1)	Release(L0)

Tabla 1: ¿Cuál es el valor consistente de X2?

para variables de este último tipo[2]. Sin embargo, es muy usual encontrar aplicaciones paralelas que modifican variables de forma irrestricta, utilizando como única herramienta de sincronización las barreras. Es por eso que la sobrecarga de ejecución de la primitiva *barrier* estará estrechamente relacionada a la vigencia de esta tercer regla, y que sólo se asumirá la validez de las reglas 1 y 2.

En la implementación de Hermes del modelo de consistencia Scope, al solicitar un nodo el *acquire* de un contexto, el pedido entra a una cola FIFO dentro del servidor del *lock* correspondiente. La concesión del *lock* conlleva el envío al nodo de todos los bloques que contienen las variables asociadas al contexto de consistencia, evitando así la ocurrencia de *misses* de cache durante la ejecución de la sesión –salvo por variables referenciadas por primera vez en tal ámbito–. El sistema maneja así una granularidad de transmisión y sincronización variable, con el bloque de cache como unidad básica, y asociada al tamaño actual del contexto de consistencia. Existen varias posibilidades para la obtención de los bloques, de acuerdo al apoyo de hardware subyacente:

Con actualización automática En este caso todas las modificaciones son inmediatamente propagadas a los nodos *homes*, por lo que durante un *acquire* el propio servidor del *lock* pide la transmisión de los bloques desde su *home* al nuevo poseedor del contexto. Necesitamos manejar una lista de *access notices* (notificaciones de acceso) indicando que bloques fueron incorporados al contexto. Durante el *release* la lista actualizada se envía al servidor de *lock*, y se invalidan todos los bloques de cache que constituyen el contexto, ya que no serán utilizados hasta un nuevo *acquire*, luego de ser modificados en sesiones del contexto en otros nodos.

Sin actualización automática En este caso los bloques de cache permanecerán en el nodo luego de efectuado el *release*, y también la lista de *access notices*. Ante un nuevo *acquire*, el nodo enviará y posteriormente invalidará los bloques que todavía permanezcan en su cache, mientras que utilizará la lista de *access notices* para rastrear aquellos bloques que hayan sido expulsados. Finalmente la lista de *access notices* se envía al nuevo usuario del contexto.

Sin lista de *access notices* Independientemente de la existencia de un hardware de actualización automática, el nodo que realiza el *release* copia todos los bloques de cache que constituyen el contexto a una locación privada de su memoria local, y luego los invalida. Cuando un nodo inicia el *acquire* del contexto, éste se le transferirá de forma monolítica. Las ventajas de este enfoque son múltiples, ya que optimizamos la utilización del substrato de comunicaciones, suprimimos la lista de *access notices*, y disminuimos la sobrecarga asociada a las operaciones de sincronización, tanto en términos de gestión del servidor de *lock*, como en cantidad de mensajes intercambiados.

La simplicidad de implementación del protocolo de consistencia en cualquiera de sus versiones, combinada con la ausencia de estructuras de datos que crezcan proporcionalmente con la cantidad

[2]En general, en el nodo *home* se hallará una versión parcialmente actualizada de la variable, en función de las expulsiones del bloque afectado de los caches de otros nodos.

de nodos o el tamaño del problema, resultan en una sobrecarga mínima en la ejecución de las operaciones de sincronización y en un potencial para la escalabilidad prácticamente irrestricto. La existencia de los contextos de consistencia permite ajustar con certeza el alcance de las operaciones de sincronización, y asumir al momento de realizar un *acquire* que las hipotéticas copias locales de las variables asociadas estarán indefectiblemente desactualizadas. Esto nos permite introducir uno de los aportes más novedosos de este trabajo, la supresión de los intervalos y *time stamps* dedicados al mantenimiento de un orden parcial de las referencias a memoria compartida.

Hermes cuenta también con ciertas operaciones adicionales, comúnmente presentes en sistemas de DSM, que sirven un doble propósito: incrementar la funcionalidad del modelo de programación, y ofrecer herramientas para optimizar la performance de las aplicaciones. La implementación de estas operaciones en software es relativamente sencilla, ya que constituyen variaciones de la mecánica del *handler* de *misses* del sistema. En primer lugar, contamos con una operación de *prefetching* explícito, similar a la planteada en la sección anterior, pero con la posibilidad de indicar el nodo origen de la información. Además, para programas que no respondan a la regla 3 mencionada más arriba, podemos recurrir a una operación *deliver* –análoga a la presente en el DASH– que propagará una modificación en un bloque de cache a un conjunto de nodos especificados. Si además podemos almacenar en los nodos receptores el autor de la modificación, esto nos resultará de suma utilidad al momento de ejecutar una *barrier*. Debemos recordar aquí que los bloques de memoria compartida pertenecientes a nodos remotos son siempre ubicados en el cache mediante operaciones de software (como la resolución de un *miss* o un *acquire*), por lo que en una estructura de datos auxiliar podemos llevar un registro de tales bloques y de información adicional relacionada. La existencia de esta estructura de datos no afectará la escalabilidad del sistema, ya que su tamaño es proporcional al tamaño del cache y no a la cantidad de procesadores ni al tamaño del problema.

Para finalizar esta sección analizaremos la implementación de la primitiva de sincronización global o *barrier*. Para variables contenidas en contextos de consistencia, el procesamiento asociado a una *barrier* consiste simplemente en propagar al *home* las modificaciones todavía pendientes y efectuar las invalidaciones necesarias. Esta tarea la efectúa sin restricciones el último nodo en realizar un *acquire*, teniendo en cuenta que ningún otro nodo tendrá copias de los bloques afectados –salvo el *home*–. Para aplicaciones donde no se observa la regla 3 anteriormente enunciada, la carencia de estructuras de datos adicionales y de estampillas de tiempo complicará el proceso, al existir variables modificadas sobre las que no se guarda información alguna. Para cada bloque compartido presente en el cache y no perteneciente a un contexto de consistencia se deberá notificar su posesión al nodo *home* correspondiente, propagando además las modificaciones. Dado que es perentorio evitar la ocurrencia de *misses* de cache debido a su atención en software, los suprimiremos aquí utilizando un protocolo de actualización. Los nodos registrarán los poseedores de bloques compartidos, y una vez terminadas las actualizaciones harán un *multicast* de la versión consistente de cada bloque a los nodos remotos que notificaron poseer una copia. Al no poder determinar una relación de orden temporal entre dichas copias, deberemos sobreescribir todo bloque potencialmente inconsistente, aún cuando el mismo no lo sea. El uso de la operación *deliver* comentada anteriormente puede reducir sensiblemente el costo de esta operación al establecer una relación causal entre las copias de una variable, evitando así sobreescrituras innecesarias.

4. Soporte de Hardware

En esta sección del trabajo detallaremos el soporte de hardware necesario para la implementación de Hermes. Como ya mencionamos en la introducción, un problema fundamental de los sistemas de DSM por software es la sobrecarga asociada con el pasaje de mensajes por métodos tradicionales. Existe una opción inicial para resolver este problema, consistente en la utilización de un procesador dedicado al tratamiento de mensajes de la red. Este procesador especial se puede

localizar dentro de la interfaz de la red (como en el Berkeley NOW), o puede ser cualquiera de los procesadores componentes de un SMP (como en el Intel Paragon). En ambos casos la mejora en los tiempos de transmisión y recepción de mensajes es sustancial [14], por lo que ambas alternativas representan soluciones válidas.

Una segunda opción consiste en adoptar una tecnología especialmente diseñada para aliviar este problema en los sistemas de DSM por software, las llamadas interfaces mapeadas a memoria virtual [3]. La idea de este tipo de interfaces es utilizar ciertas regiones de memoria virtual como canales directos a otros procesadores a través de la red, como observamos en la figura 4. Al escribir en una de estas regiones, la escritura será capturada por la interfaz y propagada automáticamente al procesador receptor –por ende el término actualización automática–. Las acciones por software vinculadas al pasaje de mensajes se reducen entonces a una simple instrucción STORE, eliminándose así las notables sobrecargas asociadas a los protocolos usuales, como TCP/IP. El costo de software para el envío de un mensaje desaparece, y la latencia –sobre todo para mensajes pequeños– se reduce exclusivamente al tiempo de transmisión de la interfaz y de la red de interconección. Contamos además con un mecanismo implícito de protección para el acceso a la interfaz de red, provisto por el subsistema de memoria virtual del procesador.

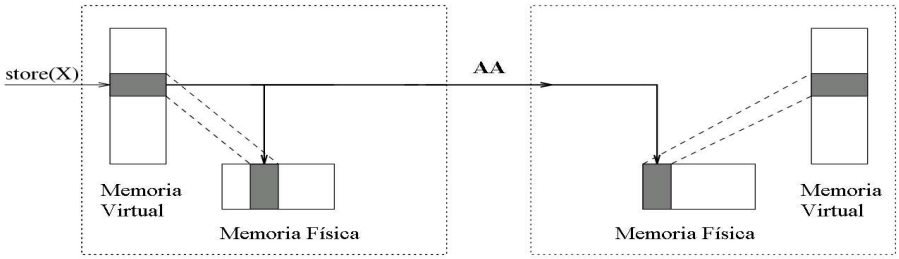

Figura 4: Mecanismo de actualización automática para interfaces mapeadas a memoria virtual.

Para Hermes, nuestra intención es utilizar la interfaz Memory Channel de DEC [8], que se puede conectar al bus PCI presente en la mayoría de las máquinas en existencia. Dentro de la tabla de páginas, asignamos a algunas páginas virtuales que serán marcadas como *salientes* una dirección de E/S dentro del bus PCI. Al escribir en una de esas páginas, la traslación enviará la escritura a través del bus PCI a la interfaz Memory Channel, y para esa dirección en particular la misma posee una tabla de traslación propia que indicará a que página de memoria física de que nodo propagarla. De la misma forma se mapean páginas de memoria virtual como *entrantes*. Las mismas deberán estar fijas en memoria (*pinned*) para recibir modificaciones remotas de páginas salientes de otros nodos, a través de la tabla de traslación de la interfaz local. La interfaz Memory Channel provee un modelo simple de actualización automática, con una performance sustancialmente superior a los métodos de pasaje de mensajes estándares La latencia mínima alcanzada por el Memory Channel para el envío de un mensaje pequeño es de 2.9 μs, contra 190 μs que consume la transmisión del mismo mensaje utilizando TCP/IP sobre Ethernet de 10Mbit.

Al integrar esta interfaz en nuestro sistema debemos diseñar un protocolo *ad hoc* que permita la transmisión de mensajes de control comunes, la transmisión de mensajes con bloques para ser cargados por la primitiva *acquire* o el *handler* de misses, y la posibilidad de realizar propagación automática de las modificaciones. Esto último se logrará alocando en cada nodo un número fijo de páginas de memoria virtual como compartidas y fijando las mismas en memoria principal. Dentro del cache virtual los bloques de nodos remotos –que están mapeados a páginas salientes– se escribirán con política *write through*. Gracias al TLB obtendremos la traslación hacia la dirección apropiada en el bus PCI para la interfaz Memory Channel, que propagará la escritura a la memoria física del nodo remoto. Deberíamos preocuparnos en principio de mantener una coherencia entre la memoria física y el cache virtual en cada nodo, pero teniendo en cuenta el modelo de consistencia planteado en este trabajo –y la implementación desarrollada–, esto no será necesario. El procesador

sólo necesita tener una visión coherente en su cache de las variables para las cuales ha abierto una sesión de un contexto de consistencia, y luego de que recibe en su cache la copia más actualizada de las mismas durante el *acquire*, nadie podrá acceder al contexto en simultáneo junto a él.

Por el momento, la implementación práctica de Hermes en un cluster de workstations se ve impedida por la inexistencia de una arquitectura comercial que cuente con una jerarquía de caches virtuales. Por ejemplo, encontramos que el cache de instrucciones L1 del Alpha 21164 cuenta con tags e índices virtuales, pero el resto de los componentes de la jerarquía son tradicionales caches físicos. La ausencia de arquitecturas que adopten los caches virtuales se explica por tres razones estrechamente relacionadas:

- Por un lado, los caches virtuales son afectados por numerosos problemas debidos a los sinónimos, aunque estos son fácil y elegantemente solucionables mediante el uso de la segmentación.
- Sin embargo, no hay espíritu en el mercado actual de microprocesadores comerciales para encarar proyectos con modificaciones tan significativas a un modelo arquitectural estandarizado. Esto ocurre por el difundido uso de los microprocesadores como *building blocks* de configuraciones SMP de mayor poder computacional y probado éxito comercial.
- Dado que los SMP se basan en protocolos de *snooping* sobre las transacciones en un bus común conectado a la memoria principal, un procesador con caches virtuales que desconozca en principio la traslación del bloque sobre el que trabaja no podrá conectarse a ese bus. Existen dos alternativas para solucionar este problema: el almacenamiento ya comentado de la traslación física para cada bloque del cache virtual, o la utilización de un bus dual con líneas para realizar *snooping* sobre la dirección virtual y líneas para comunicar la dirección física al módulo de memoria. En ambos casos está involucrado también el uso de hardware adicional no estandarizado.

El resto de los requerimientos arquitecturales para la implementación de Hermes, como la incorporación de la instrucción *Mapped Load* al *instruction set*, o un tamaño significativo de cache, están ampliamente difundidos dentro de las arquitecturas comerciales conocidas.

5. Conclusiones y Trabajo Futuro

En este trabajo hemos comentado los lineamientos sobre los que se ha desarrollado la evolución de los sistemas de DSM durante la última década, identificando características positivas de las distintas propuestas presentadas en ese tiempo. A continuación hemos propuesto un nuevo diseño al que llamamos Hermes, que apoyándose sobre el uso de una jerarquía de caches virtuales y traslación de direcciones por software, integra varias de dichas características de forma simple, y con complejidad y costo mínimo: control de granularidad fina, implementación en software, ausencia de estructuras de datos que obstaculicen la escalabilidad, y flexibilidad absoluta para la utilización de modelos de consistencia y otro parámetros del sistema. Sobre esta base hemos desarrollado una implementación del modelo de consistencia Scope Consistency para el cual se profundiza la tendencia iniciada, constituyendo en definitiva un sistema de procesamiento paralelo totalmente innovador, ágil, eficiente, altamente escalable, y carente de la significativa sobrecarga de procesamiento asociada tradicionalmente a los sistemas de DSM por software. Equipando al sistema con una interfaz de red mapeada a memoria virtual logramos el último de los objetivos planteados –la minimización sustancial de la latencia promedio para la transmisión de mensajes–, y estamos en condiciones de asegurar que Hermes constituye una alternativa barata y competitiva dentro del campo del procesamiento en paralelo. Si bien la implementación del sistema está supeditada al desarrollo de una arquitectura con caches virtuales, planeamos continuar el desarrollo de Hermes con un cuidadoso trabajo de simulación del sistema, para confirmar sus potencialidades y ajustar distintos parámetros de diseño.

Referencias

[1] C. Amza, A. L. Cox, K. Ramajamni y W. Zwaenepoel. Tradeoffs between false sharing and aggregation in software distributed shared memory. *Proc. of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'97)*, páginas 90–99, 1997.

[2] Brian N. Bershad, Matthew J. Zekauskas y Wayne A. Sawdon. The Midway distributed shared memory system. *Proceedings COMPCON '93*, páginas 528–537, 1993.

[3] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten y Jonathan Sandberg. Virtual memory mapped network interface for the SHRIMP multicomputer. *Proc. of the 21st Annual International Symposium on Computer Architecture*, páginas 142–153, Abril 1994.

[4] DeQing Chen, Chunqiang Tang, Xiangchuan Chen, Sandhya Dwarkadas y Michael L. Scott. Multi level shared state for distributed systems. *International Conference on Parallel Processing*, páginas 131–140, Agosto 2002.

[5] Fredrik Dahlgren y Josep Torrellas. Cache-only memory architectures. *IEEE transactions on Computers*, 48(6):72–80, Junio 1999.

[6] Steven Frank, Henry Burkhardt III y James Rothnie. The KSR1: Bridging the gap between shared memory and MPPs. *Proceedings COMPCON '93: Digest of Papers*, páginas 285–294, Febrero 1993.

[7] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta y John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *25 Years ISCA: Retrospectives and Reprints*, páginas 376–387, 1998.

[8] Richard B. Gillett. Memory Channel network for PCI. *IEEE Micro*, 16(1):12–18, 1996.

[9] Erik Hagersten, Anders Landin y Seif Haridi. DDM – A Cache-Only Memory Architecture. *IEEE Computer*, 25(9):44–54, 1992.

[10] L. Iftode, J. P. Singh y K. Li. Scope consistency: A bridge between Release consistency and Entry consistency. *Proc. of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA'96)*, páginas 277–287, 1996.

[11] Bruce Jacob y Trevor Mudge. Uniprocessor virtual memory without TLBs. *IEEE Transactions on Computers*, 50(5):482–499, Mayo 2001.

[12] P. Keleher, S. Dwarkadas, A. L. Cox y W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. *Proc. of the Winter 1994 USENIX Conference*, páginas 115–131, 1994.

[13] Pete Keleher, Alan L. Cox y Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *Proc. of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, páginas 13–21, 1992.

[14] Arvind Krishnamurthy, Klaus E. Schauser, Chris J. Scheiman, Randolph Wang, David E. Culler y Katherine A. Yelick. Evaluation of architectural support for global address-based communication in large-scale parallel machines. *Architectural Support for Programming Languages and Operating Systems*, páginas 37–48, Octubre 1996.

[15] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28*, (9):690–691, Septiembre 1979.

[16] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennesy, Mark Horowitz y Monica S. Lam. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, Marzo 1992.

[17] Cathy May, Ed Sikha, Rick Simpson y Hank Warren. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufman, 1994.

[18] Daniel J. Scales, Kourosh Gharachorloo y Chandramohan A. Thekkath. Shasta, a low overhead, software-only approach for supporting fine-grain shared memory. *Procs. of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, páginas 174–185, Octubre 1996.

[19] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus y David A. Wood. Fine-grain access control for distributed shared memory. *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, páginas 297–306, San Jose, California, 1994.