Hermes: DSM por software con granularidad fina

Horacio Andrés Lagar Cavilla Rafael Benjamín García

Departamento de Ciencias e Ingeniería de Computación
Universidad Nacional del Sur
Bahía Blanca, 8000
TE: (291) 4595135
FAX: (291) 4595136
{alc,rbg}@cs.uns.edu.ar

Resumen

Aún a pesar de haber pasado su etapa de mayor auge a principios de los '90, los sistemas de DSM por software pueden representar todavía una alternativa con una excelente relación costo/performance para ejecutar procesamiento en paralelo en clusters de workstation estándares, caracterizados además por un gran potencial para la escalabilidad. En Hermes enfocamos desde otro ángulo la problemática de estos sistemas, al proveer un control de granularidad fina sobre los datos con una complejidad y sobrecarga mínimas, que a la vez le brinda una alta flexibilidad al sistema para utilizar el modelo de consistencia que resulte más apropiado. Los principios que guiaron el diseño de nuestro sistema fueron dos: el de diseñar un sistema de DSM por software simple y eficiente que pudiera utilizarse sobre una plataforma estándar sin requerimientos onerosos de hardware, y el de minimizar el efecto negativo de los dos problemas principales que aquejan desde su concepción a los sistemas de DSM por software: el *false-sharing* proveniente de la gruesa granularidad de consistencia que implica el uso del mecanismo de protección de memoria virtual, y las latencias prohibitivas asociadas al tráfico de mensajes sobre protocolos de red estándares durante las etapas de sincronización. Para esto último proponemos como alternativa recurrir al uso de interfaces de red mapeadas a memoria virtual, o de un procesador adicional dedicado en el mismo nodo.

Hermes es un trabajo en progreso dentro del área de Sistemas del Departamento de Ciencias e Ingeniería de la Computación de la Universidad Nacional del Sur de Bahía Blanca, iniciado como Trabajo Final para la carrera de Ingeniería en Sistemas de Computación por Horacio Andrés Lagar Cavilla, con la supervisación del Ing. Rafael Benjamín García.

1. Introducción

Las mejoras en performance de los procesadores y redes de comunicación comerciales han permitido considerar los clusters de workstations estándares como una alternativa válida y accesible para el procesamiento en paralelo, en comparación con multiprocesadores con una similar cantidad de procesadores, mejor performance, pero una diferencia de costo sustancial y menor escalabilidad. Los sistemas de DSM (*Distributed Shared Memory* - Memoria Compartida Distribuida) intentan enmascarar la necesidad de enviar explícitamente mensajes entre los nodos componentes al proveer una ilusión de un espacio global de memoria compartida por todos los procesadores. La idea subyacente implica interceptar los accesos a datos residentes en memo-

rias remotas y de forma transparente traducirlos en mensajes sobre la red de interconexión para obtener esos datos y proveerselos a las aplicaciones. Si bien existen sistemas de DSM por hardware nuestro interés se centra en los sistemas de DSM por software que puedan ejecutarse sobre plataformas estándares con una complejidad y costo aceptables. Estos sistemas además ofrecen posibilidades de escalamiento más allá de las actualmente practicables en el ámbito de los multiprocesadores con memoria compartida.

Dos factores perjudican sustancialmente la performance de los sistemas de DSM por software, como se puede constatar en TreadMarks [**?**], punto de referencia y comparación casi obligado dentro de este tipo de sistemas. Primeramente, se utiliza el hardware de protección de memoria virtual para detectar los *misses* en la lectura de datos no actualmente presentes en el nodo local. Esta detección se hace entonces a un nivel de página de memoria virtual, con una granularidad demasiado gruesa que desencadena los indeseables efectos del *false sharing*, o invalidaciones debidas a falso compartimiento de páginas, y la fragmentación interna, o información superflua incluida dentro de las páginas que se comparten. El otro problema fundamental es la considerable latencia promedio asociada al envío de mensajes a través de protocolos de red estándares para acceder a los datos localizados en locaciones remotas.

En este trabajo proponemos un sistema de DSM por software con un control de granularidad fina sobre los datos compartidos. Este control se incorpora de forma natural a la dinámica del sistema sin incurrir en ninguna sobrecarga adicional y nos provee un método simple para implementar cualquier modelo de consistencia que consideremos apropiado. Estamos considerando utilizar el modelo de consistencia Scope, que nos brinda una interfaz de programación ya conocida, comparable a la del modelo Release Consistency, pero con una optimización de la performance y necesidad de comunicación cuasi óptima, similar a la del modelo Entry Consistency. Nuestra implementación de este modelo busca profundizar más en las ventajas inherentes al mismo, minimizando la necesidad de comunicación en los puntos de sincronización y la sobrecarga por software con una implementación sencilla y carente de estructuras de datos globales que la hacen totalmente escalable.

Por último resaltamos el hecho que un soporte de hardware, 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 interconexión, permitiendo mejorar de forma sustancial la performance de este tipo de sistemas, y potenciando su viabilidad como alternativa no sólo accesible sino eficiente para el procesamiento en paralelo.

2. Hermes

El funcionamiento de Hermes se basa en el uso de un cache de índice y tag virtuales (*cache virtual* de aquí en adelante). Un cache virtual recibe referencias a direcciones virtuales directamente desde el procesador, sin la necesidad de una traslación previa de dirección virtual a física. Dentro de los bloques del cache se indexa directamente con una porción de la dirección virtual, mientras que la otra se usa para el tag. En caso de no encontrarse el bloque deseado en el cache, recién en ese momento será necesaria una traslación de la dirección virtual a su equivalente física para referenciar dentro de la memoria principal. Con ambos niveles de cache virtual, el hit ratio se eleva por encima de un 95 %, por lo que la necesidad de realizar traslaciones se ve sensiblemente disminuida. En [**?**] se prueba exhaustivamente 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. Necesitamos entonces una arquitectura con cache virtual que genere una excepción en un *miss* de cache que pueda ser atendido por un manejador en el software (*handler* de aquí en adelante).

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 del 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 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 ese bloque (con N procesadores). Para datos locales el *handler* recorre la tabla de páginas y obtiene el PTE de la página donde reside el dato buscado. Mediante la adición al set de instrucciones de la instrucción *mapped load* (carga mapeada) de dos operandos, se puede ubicar un bloque determinado en el cache virtual. Uno de los operandos especificará la dirección virtual y bits adicionales del bloque, y el otro indicará la dirección física desde donde se realizará la carga, extraída del PTE correspondiente. Para una referencia remota el *handler* puede directamente emitir el mensaje con el pedido del bloque para luego cargarlo en el cache de manera similar.

Extendiendo de esta forma una componente del sistema ya presente, podemos integrar de manera simple y efectiva un control de granularidad fina (la unidad de granularidad sería el tamaño del bloque de cache L2) para nuestro sistema de DSM. De esta manera reducimos considerablemente el problema del 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. Esta semejanza con las arquitecturas COMA tradicionales implica disponer de la colocación de los datos en memoria cuidadosamente para evitar la posibilidad de que dos nodos simultáneamente escriban a variables distintas dentro de un mismo bloque (y alguna de las modificaciones se pierda al escribirse en el nodo *home*). Además, el manejo por software de los misses permite implementar cualquier tipo de tabla de páginas virtuales que consideremos apropiada y prácticamente cualquier modelo de consistencia, brindando así una gran flexibilidad al sistema.

La utilización de un esquema segmentado como el presente en el PowerPC permite manejar inócuamente varios de los problemas que afectan a las arquitecturas con cache 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. Evitamos de esta forma algunos problemas clásicos de los cache virtuales, como los sinónimos, y las invalidaciones en cambios de mapeos y cambios de contexto. Sin embargo, la segmentación no es una solución para todos los problemas de los cache virtuales. Si deseamos mantener bits de protección a un nivel de página de memoria virtual, deberemos replicar estos bits para todos los bloques de cache de una página, y en caso de una modificación recorrer el cache para actualizar el estado de los bits. Sostenemos que en un ambiente DSM la frecuencia relativamente baja de cambios de permisos de protección disminuirá el efecto de esta costosa operación.

El diseño original de [**?**] propugnaba eliminar el TLB (al volverse éste innecesario) y liberar así espacio en chip. Bajo este escenario, y con un cache utilizando política *write-back*, es razonable suponer que la mitad de los reemplazos por *misses* implicarán la expulsión de un bloque del cache y la subsecuente traslación por software de la dirección virtual del bloque para su escritura. El TLB es un componente presente en todas las arquitecturas estándares, y proponemos utilizarlo para manejar las traslaciones de páginas con permiso de escritura y con al menos un bloque presente en el cache virtual, y evitarnos así el problema planteado en las escrituras.

Requerimos como condición instrumental que el tamaño del cache virtual sea sustan-

cial, en el orden de 4 u 8 MB para el cache L2. Esto es necesario por dos motivos: el esquema de manejo de *misses* por software requiere un *hit ratio* alto para ofrecer una performance competitiva, y el hecho de alojar bloques remotos exclusivamente en el cache (en un estilo COMA) hace que un buen tamaño del mismo permita minimizar la necesidad de costosos accesos remotos.

3. Modelo de consistencia Scope

El modelo de consistencia Scope [**?**] que utilizaremos se basa en el uso de contextos de consistencia que encapsularán un conjunto de variables cuyo acceso debe ser sincronizado. Mediante el uso de operaciones *acquire* y *release* sobre el *lock* correspondiente arbitramos el acceso a las variables contenidas en el contexto de consistencia asociado a dicho *lock*. Llamamos sesión de un contexto a la porción del programa donde se realiza el *acquire* del *lock*, 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 por asunciones que ya no son válidas (referirse a los autores[**?**]), este comportamiento ofrece una performance similar al de la consistencia Entry, focalizando el gasto en actividades de sincronización en las variables que exclusivamente lo necesitan y minimizando de esta manera la necesidad de comunicación. Aparte de las primitivas *acquire* y *release* proveemos la tradicional primitiva *barrier* y una herramienta para crear dinámicamente nuevos contextos, la primitiva *create_scope*.

El protocolo de Hermes para implementar la consistencia Scope utilizará los bloques de cache como unidad básica de granularidad. Al solicitar un *acquire* el pedido entra a una cola FIFO dentro del servidor del lock correspondiente. Cuando se le concede el *lock* se le envían al nodo todos los bloques que contienen las variables que constituyen actualmente el contexto de consistencia. Estos bloques se pueden obtener de una lista de *access notices* asociados a cada *lock*. Manejamos dos posibilidades, de acuerdo al apoyo de hardware subyacente: o los bloques provienen de sus nodos *home*, en un ambiente con actualización automática, o los bloques le son enviados directamente desde el nodo que abrió la anterior sesión del contexto y por lo tanto fue el último en modificarlo. Podemos decir de esta forma que tenemos una granularidad de transmisión y sincronización variable asociada al tamaño actual del contexto de consistencia. Además, las únicas estructuras de datos asociadas al protocolo son las colas FIFO y las listas de *access notices* asociadas a los *locks*. La simpleza de esta implementación permite una sobrecarga mínima en la ejecución del protocolo, y además la escalabilidad del sistema no se ve comprometida en absoluto. Sin embargo, no hemos mesurado todavía el impacto que esto puede tener en la ejecución de las *barriers*.

4. Soporte de Hardware

Al describir Hermes hemos señalado algunos requerimientos elementales de hardware para la implementación del mismo: cache virtual, excepciones manejables por software en el caso de *misses*, posibilidad de implementar la instrucción *mapped load*, segmentado, y un tamaño importante de cache. Una arquitectura comercial como la del PowerPC de IBM cumpliría todos estos requisitos.

En cuanto a la sobrecarga asociada con el pasaje de mensajes por métodos tradicionales, amén de la posibilidad ampliamente explorada de utilizar un procesador adicional dedicado, analizaremos aquí el uso de interfaces mapeadas a memoria para atacar este problema. Nuestra intención es utilizar la interfaz *Memory Channel* de DEC para Hermes.

La misma se puede "colgar" del 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 I/O dentro del bus PCI. Al escribir en una de esas páginas, la traslación enviará la escritura a través del bus PCI a la interfaz, y para esa dirección en particular la misma posee una tabla de traslación que indicará a que página de memoria física de que nodo propagar la modificación. 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. Las acciones por software vinculadas al pasaje de mensajes se reducen entonces a una simple instrucción STORE que inmediatamente se verá implementada en el nodo destino, y por eso se llama a este mecanismo *actualización automática*. Eliminamos así las notables sobrecargas asociadas a los protocolos usuales, como TCP/IP, y la latencia en el envío sobre todo de mensajes pequeños se reduce prácticamente al tiempo de transmisión de la interfaz y red.

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 caché virtual los bloques de nodos remotos se escribirán con política *write through* y gracias al TLB obtendremos la traslación hacia la dirección en el bus PCI de una página saliente. El Memory Channel propagará la escritura desde la página saliente a la memoria física del nodo remoto. Ciertas precauciones deben ser tenidas en cuenta relacionadas con el modelo de consistencia, ya que no podemos manejar una coherencia por hardware entre memoria física y cache virtual en el nodo destino.

5. Trabajo Futuro

Si bien el diseño integral del sistema no ha sido completado, tenemos varias opciones abiertas que sólo podremos decidir después de una cuidadosa simulación de la performance de nuestro sistema:

- Organización y algoritmo de recorrido de la tabla de páginas virtuales.
- Optimizaciones varias para el *handler* de *misses* por software.
- Tamaño de cache óptimo para la performance del sistema.
- Alternativas para el uso del Memory Channel con políticas *write back* o *write through*.
- Uso extensivo de *prefetching* de bloques remotos, tanto en el momento del *miss* como a través de pedidos explícitos.
- Implementación de *barriers*, con o sin el soporte del Memory Channel.
- Optimizaciones del protocolo para el caso usual de los *properly labelled programs*.