Protocolos Especulativos: Hacia una Eficiente Implementación en Software

Jorge Pérez Jaime Vásquez
e-mail: japerezr@puc.cl T: 02-6354929 F: 02- 6864444
Departamento de Ciencia de la Computación, Escuela de Ingeniería
Pontificia Universidad Católica de Chile

Resumen

La memoria compartida distribuida (DSM) está cada día siendo más utilizada, debido a sus ventajas por sobre una memoria convencional y centralizada, principalmente precio y escalabilidad. Sin embargo la memoria compartida distribuida tiene sus desventajas y problemas, los cuales se derivan principalmente de la latencia introducida al transferir páginas de memoria desde un nodo a otro. Una manera de disminuir esta latencia y, por consiguiente, mejorar el desempeño del sistema sería predecir con antelación las futuras operaciones de consistencia y las peticiones de una página en remota, realizando todo el proceso anticipadamente. En esta área se han realizado algunos trabajos, pero estos abarcan sólo implementaciones en hardware. El objetivo de este artículo es ver la posibilidad de, basándonos en los trabajos anteriores enfocados a hardware y en el sistema de memoria compartida distribuida PEPE, proponer formas de implementar predicción en software, planteando una heurística aplicable sobre PEPE, que pudiera ser el punto de partida para llegar más adelante a un sistema de este tipo que mejore la eficiencia de un DSM sin requerir hardware especial.

1. Introducción

La creciente demanda por aumentar la eficiencia y respuesta de grandes aplicaciones computacionales, ha incentivado la creación de arquitecturas de computación paralela. En general una mayor rapidez conlleva una mayor dificultad en lo que a la programación de usuario se refiere. Los Sistemas de Memoria Compartida Distribuida (DSM) alcanzan un equilibrio entre escalabilidad y simplicidad en la programación de aplicaciones paralelas, implementando un espacio global de direcciones compartidas montado sobre varias máquinas uni o multiprocesador.

Dado que los sistemas DSM ofrecen transparencia en la interfaz, un modelo conveniente de programación y compatibilidad con el software paralelo existente, están ganando amplia aceptación tanto en ambientes técnicos como comerciales.

Por las características anteriores, diversos estudios han intentado aumentar aún más la eficiencia de los DSM. Las opciones de optimización son muchas. En este artículo nos centramos en el aumento de la eficiencia mediante la distribución especulativa de datos en los procesadores. Diversos trabajos han tocado el tema siempre desde el punto de vista de las implementaciones en hardware, ahora proponemos ideas y métodos para implementaciones especulativas en software. La mayoría de los ejemplos son fácilmente aplicables en sistemas comunes y usamos el sistema PEPE [4] como base para la aplicación de nuestros métodos.

En la siguiente sección describimos algunos métodos de predicción que aparecen en la literatura. En la sección 3 analizamos algunos aspectos que nos parecen serían los más relevantes al momento de intentar implementar protocolos especulativos por software. La sección 4 describe brevemente el funcionamiento del sistema PEPE sobre el cual se ejemplificarán los métodos propuestos. En la sección 5 proponemos una nueva heurística que llamamos *Heurística Especulativa Write Done*. Finalmente la sección 6 presenta las conclusiones y trabajo futuro.

2. Algoritmos de Predicción

Los principales problemas de escalabilidad y desempeño de los sistemas DSM se deben en gran medida a que, en numerosas ocasiones, los nodos que conforman el sistema deben hacer acceso a memoria remota, es decir, acceso a datos que no se encuentran en su memoria local ni en su caché, o de encontrarse en éste están en un estado inválido. El acceso a memoria remota es considerablemente más lento que un acceso a memoria local (llegando a tardar hasta 100 veces más [1]), y en cada acceso se debe pagar un costo considerable por las acciones de sincronización

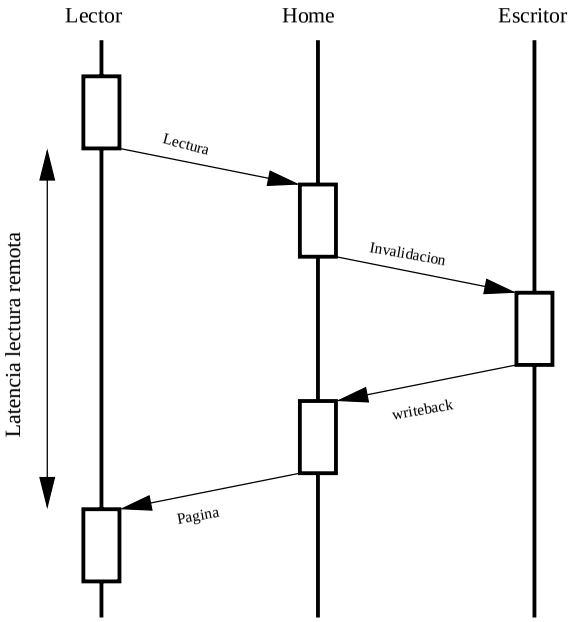

Figura 1: **Lectura remota a una página en un sistema basado en homes. La lectura significó cuatro mensajes por la red.**

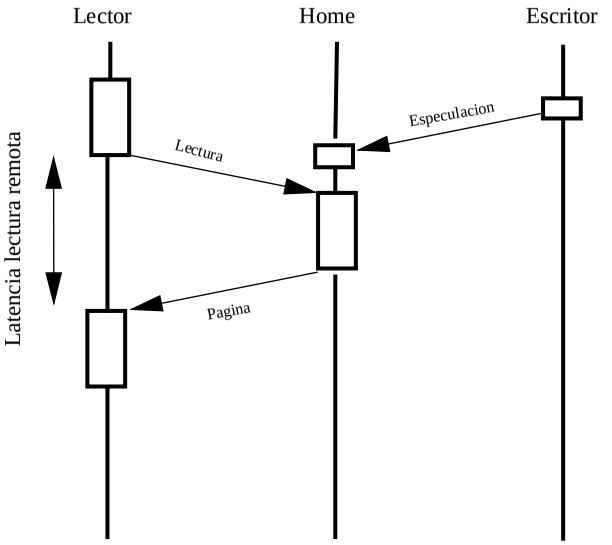

Figura 2: **Especulación por parte del nodo Escritor, devuelve la página a su Home. La siguiente lectura disminuye su latencia a la mitad.**

que se deban realizar para mantener la coherencia en los sistemas de caché del resto de los nodos. El costo de un acceso no local puede variar dependiendo del modelo de consistencia que se adopte en la implementación, pero sea cual sea el elegido, hay un gasto inevitable.

Por ejemplo, en una implementación en hardware con un protocolo de un escritor y múltiples lectores con directorios basado en invalidación, cuando un nodo desea leer una página que no se encuentra en memoria local (o está en estado inválido en su memoria caché), debe enviar un mensaje al nodo HOME de la página solicitando una copia. Si este se da cuenta de que la página en ese momento está siendo escrita por algún otro nodo, debe enviar un mensaje de invalidación a este último para obtener la página de regreso, y sólo en ese momento puede enviar una copia al nodo que en un principio solicitó la lectura. La lectura entonces incluyó cuatro mensajes por la red. En este caso se ve claramente la gran latencia de una lectura remota comparada con una lectura sobre la memoria local, el procesador debió esperar a que se transmitieran y fueran procesados cuatro mensajes por la red antes de poder continuar su ejecución. Un esquema de este proceso se puede ver en la Figura 1.

Por las razones anteriores diversos investigadores se han abocado a la tarea de proponer algoritmos y protocolos que intenten predecir las operaciones de coherencia y acceso remoto en los sistemas de memoria compartida distribuida, de manera de reducir la latencia y aumentar la eficiencia del sistema DSM, ejecutando por adelantado las operaciones de sincronización necesarias. Por ejemplo en la Figura 1, si el hardware del sistema DSM en el nodo Escritor predice efectivamente que se producirá una lectura por parte del nodo Lector, podría entonces invalidar la página y enviársela al nodo Home por adelantado. Una subsiguiente lectura a la página la encontrará actualizada en el nodo Home, reduciendo así la latencia de lectura a la mitad, como se puede ver en la Figura 2. Un caso muy favorable sería que además el nodo Home predijera la lectura del nodo Lector y le enviara la página actualizada por adelantado, escondiendo así completamente la latencia de lectura convirtiéndola en un acceso local, Figura 3.

Los algoritmos existentes, fruto de las investigaciones en el tema, utilizan diversas heurísticas para implementar la predicción en sistemas DSM, las cuales están enfocadas principalmente a sistemas de hardware. Por ejemplo en [2] se utiliza la historia reciente de accesos a memoria y mensajes de sincronización. Esta información es almacenada en tablas, en las cuales se conserva el orden de las operaciones de sincronización ocurridas, a partir de estas tablas se identifican secuencias o patrones que caracterizan grupos de operaciones que generalmente se ejecutan seguidas, así cuando se detecta el inicio de alguno de estos patrones se ejecuta todo el resto, asumiendo que de todas formas se ejecutarán más adelante. Este algoritmo necesita entonces de hardware especial

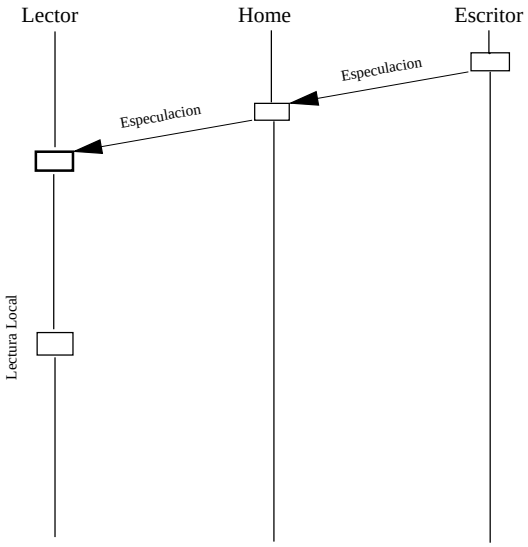

Figura 3: **Doble especulación por parte de los nodos Escritor y Home. Se esconde por completo la latencia de lectura remota.**

para poder almacenar estas tablas, las cuales aumentan la tasa de predicciones acertadas a medida que se aumenta la cantidad de información que puedan contener. Esto último hace que, si se quieren muy buenos rendimientos (en cuanto a la tasa de aciertos), se deba aumentar prohibitivamente el tamaño de las tablas y por ende el costo del sistema.

En [3] encontramos una forma novedosa de atacar el problema. Se trata de disminuir el overhead de sincronización prediciendo cuando un procesador ha hecho su última escritura sobre una página determinada (*Last-Touch Prediction*), pudiendo después de esto, autoinvalidarse y devolver esta página a su nodo HOME para que futuras lecturas a ella la encuentren actualizada. Esto ahorraría dos mensajes en la red y disminuiría, en caso de acierto, la latencia por lectura a la mitad, Figura 2. Las implementaciones que se proponen para esta idea, funcionan para protocolos de invalidación con un escritor y montados sobre sistemas con hardware de sincronización que es posible aprovechar para el algoritmo casi sin requerir modificaciones. De todas formas este algoritmo requiere tener acceso a todas las instrucciones de lectura y escritura que haga un procesador sobre una dirección de memoria en particular para poder operar. Es importante tomar en cuenta que una falla en la predicción en un sistema que implementa un algoritmo como este puede resultar muy perjudicial, ya que convertiría un futuro acceso que debió ser local en remoto.

En general, estos sistemas han mostrado un buen resultado en simulaciones y su comparación con otros métodos para aumentar el rendimiento ha sido favorable. Una de las principales ventajas que tienen, es el que en general son métodos simples de aplicar si se cuenta con el hardware que lo permita, a diferencia de optimizadores en tiempo de compilación que utilizan complicados algoritmos, métodos que necesitan de mucha ayuda del programador, o relajaciones a los modelos y protocolos de consistencia. Quizá la más favorable de sus características es que no se necesita cambiar nada del protocolo de sincronización en uso, sino simplemente, pedirle al administrador DSM que realice una operación de sincronización por adelantado. En general todos los aprontes al tema, tal como se nota en los dos métodos mencionados anteriormente, requieren de hardware especial o en el mejor de los casos aprovechan ciertas características presentes en hardware de sincronización más común, el cual de todas formas debe cumplir ciertas características mínimas. Solamente se ha comentado de forma muy superficial la utilización de métodos de este tipo mediante software, esto debido principalmente a que se tiende a pensar que un sistema de este tipo produciría un overhead inaceptable, y no se han hecho estudios que analicen en profundidad los problemas a resolver para conseguir una implementación de estos algoritmos en software que produzca una mejora en el desempeño de un DSM.

3. Predicción en software

A continuación describiremos algunas características que no han sido estudiadas en profundidad en los sistemas actuales y que nos parece que podrían mejorar los mecanismos de predicción, además intentaremos identificar algunas de sus falencias, para luego enfocarnos en identificar los aspectos fundamentales que debemos considerar al momento de implementar un sistema de este tipo en software.

En primer lugar es importante notar que en todos los sistemas que hemos estudiado la información almacenada y utilizada para realizar las predicciones es propia de cada ejecución y no se mantiene durante sucesivas ejecuciones. En este sentido estos sistemas funcionan más como una memoria caché en el que en cada ejecución se parte sin información para predecir, y por ende no se puede predecir hasta haber recabado la suficiente información. Consideramos que sería tremendamente útil el ser capaces de almacenar la información necesaria para realizar las predicciones entre una y otra ejecución, ya que aparte de permitir evitar la baja eficiencia del algoritmo en un principio (lo que puede no tener demasiada relevancia) podría permitir a largo plazo mejorar la calidad de las predicciones mediante el análisis y medición de la

efectividad de cada método. Esto sin embargo tendría verdadera utilidad práctica en sistemas que ejecutan más o menos la o las mismas aplicaciones a través del tiempo y que además reciben más o menos las mismas entradas o datos que procesar.

En segundo lugar la mayoría de los sistemas actuales basa sus predicciones en la información correspondiente a accesos a memoria y operaciones de sincronización. Esto lo hacen porque en estos sistemas es importante delimitar muy bien el tipo de información a almacenar debido al costo involucrado y la información mencionada es la más relevante al momento de hacer una predicción. Sin embargo seria útil analizar la posibilidad de considerar la relevancia de otros datos que pueden tener influencia en el sistema DSM, tales como el estado en general del sistema: cantidad de procesos, espacio en disco, memoria disponible, carga en cada componente, tipo de procesos en ejecución, etc. Todo lo anterior tiene influencia en el funcionamiento de un DSM y aunque considerar todos los otros parámetros constantemente es demasiado costoso, quizás sería factible hacer estimaciones cada cierto tiempo o en momentos en que el sistema dispusiera de recursos libres suficientes. Analizaremos más este tema en el momento en que propongamos nuestras mejoras a los sistemas actuales y las características de un sistema de este tipo por software.

Otro aspecto relacionado con el anterior es la posibilidad de además considerar información gatillada desde otros nodos del DSM, aparte de la información correspondiente a sincronización y acceso a paginas, esto de manera de considerar todos los aspectos que realmente influyen en el DSM, ya que la predicción dependerá siempre del estado del sistema completo aunque siempre existirán aspectos más o menos relevantes.

Finalmente sería bueno analizar la posibilidad de ir mejorando con el tiempo la calidad de las predicciones, aun cuando esto a simple vista puede llevar a un gran overhead. Sería práctico que sistemas de este tipo consideraran mecanismos en los que evaluaran su desempeño, en este caso su tasa de aciertos en la predicción, y realizaran automodificaciones, ya sea a la información que poseen o directamente en sus algoritmos. De esta manera podríamos tener sistemas que aprendieran y se adaptaran a las condiciones, mejorando su eficiencia con el tiempo, siempre y cuando también tomen en cuenta que hay que mantener los datos a través del tiempo, como mencionamos en un párrafo anterior.

¿Qué aspectos hemos identificado como fundamentales para tomar en consideración al implementar en software sistemas de predicción? A continuación enumeramos los aspectos que a nuestro juicio son los más relevantes:

1. Un sistema en software debe monitorear sólo lo justo y necesario, e idealmente no durante todo el tiempo en que esté funcionando el sistema. Esto principalmente para evitar gastar demasiados recursos en el sistema de predicción disminuyendo excesivamente la eficiencia del sistema.

2. Por lo anterior debe restringir mucho más el tipo de parámetros a observar, y en este sentido lo óptimo es que divida en grupos de parámetros, a observar dando prioridad a la observación de los más relevantes para la predicción, y observando con menor frecuencia, o descartando otros si es necesario.

3. En lo posible se debe intentar utilizar características del Sistema Operativo particular sobre el cual se trabaja para realizar el monitoreo, funciones que en general ya se encuentran optimizadas de manera de no re-implementar sistemas de monitoreo.

4. Debido a que se trabaja con software será más sencillo, aunque más costoso en términos de utilización de recursos, implementar algoritmos más inteligentes para realizar las predicciones. Esto en caso del hardware sería posible, pero encarecería inmediatamente el sistema. Es importante que de implementar algoritmos complejos se analice muy bien la cantidad de recursos utilizados, de manera que valga la pena la o las mejoras en los algoritmos.

5. También es importante considerar que en un computador en general siempre hay intervalos en que la utilización de los recursos, tales como la CPU, discos o ancho de banda en la red, tienen una menor utilización. Sería ideal que un sistema de predicción fuera capaz de distribuir sus tareas de manera que se asignen mayor cantidad de tareas según la utilización actual del sistema, aprovechando los recursos ociosos. Es importante recalcar que en este aspecto habrá operaciones del sistema que deberán ser realizadas en periodos de mayor carga, tal como el monitoreo de ciertos parámetros, ya que el comportamiento del sistema a predecir no será igual en periodos de alto o baja carga y es importante ser capaz de predecir en cualquiera de los dos casos.

6. Finalmente sería práctico aprovechar el almacenamiento estable que se posee (discos) para almacenar la información recabada en el tiempo, lo que en general no se hace con los sistemas por hardware.

Nos parece que métodos como el propuesto en [3], son poco factibles de implementar en sistemas de software debido al masivo uso de monitoreo de las lecturas y escrituras de memoria por parte de cada procesador, hacer una interrupción por cada acceso a memoria resultaría catastrófico en el desempeño. En sistemas de hardware en cambio, esto no significa un gran costo adicional. Tal vez ideas del estilo de [2] pueden ser más aplicables en software.

4. El sistema PEPE

En esta sección describiremos el sistema de memoria compartida distribuida denominado PEPE, con motivo de explicar nuestro trabajo ejemplificando sobre este sistema DSM en particular.

El sistema PEPE, es una implementación de un DSM diseñado mediante orientación a objetos sobre C++, pensado para ser usado en una red de computadores. Implementa un protocolo de sólo un escritor y por ahora soporta sólo consistencia secuencial, en donde cada página posee un propietario variable y cada nodo conoce un propietario probable que se va actualizando a medida que se realizan accesos remotos. De esta manera cuando un nodo intenta acceder a una página remota pide una copia al probable propietario, si este no la tiene avisará al primer nodo a quien se la entregó el cual pasará a ser el probable propietario, de esta manera se terminará contactando al nodo que posee la página requerida, en el peor caso preguntando a todos los nodos. En este sistema, sin embargo, el peor caso es muy improbable, usualmente la página requerida se encuentra muy rápido.

El propietario de una página de memoria cualquiera cambia con cada petición de escritura, al mismo tiempo se invalidan las copias que otros nodos tengan de esa página para lectura. Así una siguiente lectura de la página resultará en un acceso de memoria remoto. La petición de lectura siempre será hecha al nodo propietario de la página por lo que siempre se encontrará actualizada.

La implementación de Pepe, utiliza un objeto administrador de páginas, y cada página es un objeto. Esto es muy práctico si requerimos almacenar datos extra específicos de cada página, simplemente derivamos una nueva clase página con los campos extra necesarios para guardar la información.

En definitiva PEPE es un sistema DSM con un funcionamiento simple de entender, lo que lo hace indicado para ilustrar las probables ventajas de algoritmos que se puedan proponer y las mejoras a los sistemas de predicción que presentaremos.

En la siguiente sección describiremos una heurística muy simple para predicción en software orientada principalmente a mejorar el desempeño de un sistema como PEPE.

5. Heurística Especulativa Write Done (SWDH)

En este artículo proponemos un nuevo método para realizar predicciones en sistemas DSM en software del estilo de PEPE, que implementen consistencia secuencial, llamada Heurística Especulativa Write Done. El nombre se debe a que en general la idea será especular acerca del término del uso para escritura de cierta variable[1] por parte de un nodo "Productor". El funcionamiento de la heurística está basado en la observación de que el programador debe sincronizar el acceso a variables compartidas si es que quiere estar seguro de que no se producirán competencias por datos (*data races* [5]) en la ejecución sin importar el modelo de consistencia sobre el que funcione el DSM, y que muchas aplicaciones tienen un comportamiento repetitivo en cuanto a la compartición de datos entre procesadores.

La motivación para la heurística es un simple ejemplo, la conocida aplicación *El Juego de la Vida* que intenta modelar la evolución temporal de una población. La idea de la aplicación es que el nacimiento o supervivencia de un individuo de una generación a otra, está determinada por el número de individuos vecinos. La implementación típica es asignando cada lugar del espacio donde los individuos viven a un procesador y sincronizando los procesadores mediante barreras. Así la implementación se divide en dos fases, una en la que todos los procesadores leen y otra en la que todos escriben según las lecturas hechas en la fase previa. El protocolo que implementa PEPE para la consistencia de la memoria, hace que luego del término de la fase de escritura cada procesador sea dueño de la variable correspondiente al estado del individuo en la posición que le corresponde. La fase de lectura siguiente significará que cada procesador deba hacer un acceso remoto a las variables que corresponden a posiciones vecinas, produciendo un gran número de mensajes por la red, e introduciendo una gran latencia en cuanto a las lecturas. Nos interesa entonces intentar predecir de alguna forma que luego de la fase de escritura se realizarán varias lecturas y poder tomar acciones especulativamente por adelantado.

Como vemos en el caso del *Juego de la Vida* la implementación tiene un comportamiento repetitivo y

[1] En la explicación y desarrollo de SWDH se asume que la unidad de compartición entre procesadores son variables. Un análisis similar se puede llevar a cabo si se supone que los nodos comparten páginas.

además los accesos a variables que puedan producir competencia por datos están claramente sincronizados, por lo que este ejemplo cumple con las condiciones que planteamos debían ser necesarias para la aplicación de SWDH.

La heurística relacionará especulativamente una variable compartida con una variable de sincronización y tomará las decisiones dependiendo de parámetros históricos de compartición de datos de la aplicación específica. Hemos desarrollado distintos algoritmos basados en la misma idea para distintos tipos de variables de sincronización. A continuación detallaremos su uso para *locks*, *barreras* y un pequeño análisis de lo efectivo que puede ser en el rendimiento.

5.1. SWDH: *Locks*

Los *locks* son variables de sincronización que se usan en general para proveer acceso excluyente a *secciones críticas* de programas que comparten variables, ellos protegen el acceso a ciertas porciones del código mediante operaciones Aq y Re. Dado una variable L definida como lock, sobre ella un procesador puede aplicar la operación $Aq(L)$ para obtener acceso excluyente y $Re(L)$ para permitir que otro procesador pueda acceder a una sección protegida por el mismo L.

Para nuestro desarrollo tomaremos el caso en que dos procesadores A y B se "pelean" una variable X para escritura y supondremos que los accesos a X están protegidos por el lock L. Entonces si un procesador A quiere acceder a la variable X deberá antes hacer $Aq(L)$, posteriormente podrá hacer $w(X)$ que representa la escritura sobre X, y finalmente hacer $Re(L)$ permitiendo que otro procesador pueda tomar L. Si una aplicación similar ocurre bajo el sistema PEPE, luego de $w(X)$ la variable X será propiedad de A quedando todas las copias de X que tengan otros procesadores, en estado inválido.

Supongamos que luego de que A hace $Re(L)$ el procesador B quiere ingresar a su sección crítica, para lo que hace $Aq(L)$, luego realiza $w(X)$ que significa un acceso remoto porque el propietario de X en ese momento es A. Para esto B envía un mensaje al procesador A pidiéndole la propiedad de la variable X (además se realizan mensajes de invalidación de la variable a procesadores que en ese momento tengan copia de lectura, pero esto no importa en el caso que estamos analizando). Sólo después de la recepción de la propiedad, B puede escribir sobre X. Los mensajes incluyeron una latencia en el acceso a X por parte de B. Finalmente B puede hacer $Re(L)$ terminando el ciclo de escrituras. Lo anterior se puede ver en la parte superior de la Figura 4 (en ella además están identificadas las peticiones de Aq que también serán parte de la heurística y se explicarán más adelante).

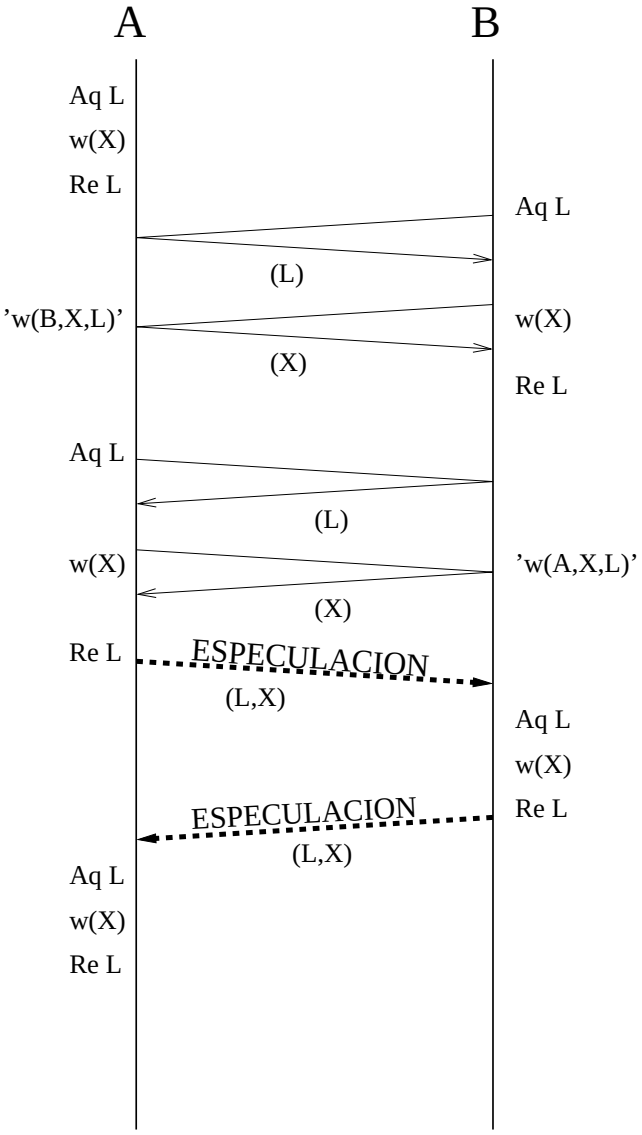

Figura 4: **Especulación en el envío de L y la propiedad de X por parte de los nodos A y B. En el mejor de los casos se esconderá completamente las latencias del $Aq(L)$ y acceso de escritura a X.**

Nuestro algoritmo propone que al momento en que A recibe el mensaje desde B para traspasar la propiedad de X, en A se almacene la información $w(B, X, L)$ la que en la Figura 4 aparece entre comillas simples. El sentido de esta información es que A recuerde que llegó una petición de escritura por la variable X, que en ese momento era de su propiedad, por parte del procesador B, el que se encontraba sincronizado por el lock L. De esta forma se relacio-

nará "especulativamente" a la variable X con el lock L. Si luego el nodo A ejecuta nuevamente su sección crítica protegida por L, la memoria $w(B, X, L)$ que tiene en ese momento le aconsejará que al momento de hacer $Re(L)$ se le envíe especulativamente la propiedad de la variable X al procesador B. Si luego el procesador B ejecuta su sección crítica, el acceso $w(X)$ que hace será local y no habrá latencia alguna.

Como la aplicación es simétrica para los dos procesadores, si B también ejecuta el algoritmo antes descrito, luego del segundo acceso de A a su sección crítica B tendrá en su memoria a $w(A, X, L)$ y una posterior operación $Re(L)$ aconsejará el envío especulativo de la propiedad de X al procesador A. En el mejor caso todas las operaciones $w(X)$ en adelante no necesitarán de mensajes por la red de petición de propiedad.

Algo que no hemos mencionado es que, dependiendo de la implementación, los locks pueden verse como un caso especial de variables compartidas. La implementación actual de locks en PEPE es centralizada, pero se están estudiando formas de poder implementarlos de una manera distribuida. Una posible implementación es tratar el lock como un *token*, en que el que posee el lock en cierto momento es al que se le permite el acceso a su sección crítica. Una implementación como esta significaría que cada petición de Aq de un lock por parte de un procesador al que no le pertenece, gatille mensajes por la red y una subsiguiente latencia. Nuestro algoritmo se puede usar también como una forma de pasar el lock especulativamente de un procesador que hace Re a otro que probablemente haga Aq, escondiendo así también la latencia en la petición del lock. Este es el caso general que se muestra en la Figura 4. Luego de que tanto el procesador A como B han ingresado los datos, que en la figura aparecen entre comillas simples, a su memoria, la especulación al momento del $Re(L)$ envía no sólo la propiedad de la variable al otro procesador si no que también el lock L, escondiendo así por completo la latencia que se introducía por el $Aq(L)$ y por el acceso a X.

Para la aplicación del algoritmo antes descrito, es necesario que cada procesador que realice una operación de sincronización de Aq, recuerde sobre que variable de lock lo hizo, para que en el mensaje que envíe pidiendo a otro procesador la propiedad de una variable se pueda guardar la información necesaria para la especulación. Es importante también que el procesador que hace Re tenga claro si puede o no realizar una especulación. Por ejemplo, si A tiene en su memoria a $w(B, X, L)$ y hace un $Re(L)$, puede enviar a B la propiedad de X sólo si esta se encuentra con permiso de escritura en A (o sea que A sea el propietario), de otra forma no debería realizar especulación alguna.

5.2. SWDH: *Barreras*

Las *barreras* son variables que se usan en general para sincronizar varios procesadores. Un procesador puede pasar la barrera sólo cuando todos los otros ya han llegado a la barrera también. Las aplicaciones que usan barreras, en general tienen comportamientos definidos en fases, cada fase la separa una sincronización de barrera.

Supongamos que tenemos una aplicación tipo *Juego de la Vida*, en que tres procesadores A, B, C, escriben sobre variables X, Y, Z, respectivamente, y en que cada escritura dependerá del valor de las tres variables. Supongamos además que las escrituras y lecturas están sincronizadas por una barrera L, es decir, los procesadores leen las variables, esperan en L, cuando todos llegan escriben sobre su variable respectiva, esperan en L y repiten el proceso. Nuestro algoritmo en este caso relacionará variables compartidas con la variable de barrera.

Para explicar el funcionamiento, supongamos que cada procesador escribe sobre la variable asignada, luego de lo cual cada uno tendrá la propiedad de la variable. Luego son sincronizados por la barrera L. Al seguir ejecutando cada uno lee la variable de la que tiene permiso de escritura y las dos variables de cada uno de los otros procesadores. Cada una de estas últimas lecturas gatillará mensajes por la red, ya que deberán leer datos que han sido modificados por otros procesadores (se debe recordar que estamos bajo consistencia secuencial). Al igual que en el caso de locks, podemos guardar cierta memoria de estos accesos. En la Figura 5 se ve un esquema del método. Después de la primera sincronización por la barrera L, el procesador A debe hacer $r(X)$, $r(Y)$, $r(Z)$ (lectura de las tres variables), la primera es una lectura local y las dos últimas son lecturas remotas, una con un mensaje hacia B y otra hacia C. Con cada uno de estos mensajes se guarda en el procesador al que llegó, los datos del acceso. Por ejemplo cuando A hace $r(Y)$, envía un mensaje a B y este dejará almacenado $r(A, Y, L)$ que significa que recibió una petición de lectura de la variable Y por parte del procesador A que estaba sincronizado por la barrera L. Análogamente se almacenará $r(A, Z, L)$ en el procesador C. La simetría de la aplicación hace que los tres procesadores tengan almacenada información del los accesos antes de llegar a la siguiente sincronización de barrera.

Luego de pasar la barrera no se efectúa ninguna especulación ya que ningún nodo tendrá permiso de es-

critura sobre ninguna página. Ahora, cada nodo volverá a escribir sobre su correspondiente variable. El escribir significará solo un overhead en la red, necesario para invalidar todas las copias que los otros procesadores tengan con permiso de lectura de la variable que un procesador en particular va a modificar. En la figura este paso está marcado por una serie de mensajes etiquetados por "invalidación".

Al llegar todos lo procesadores nuevamente a la barrera L, se darán cuenta de que pueden realizar una especulación. Por ejemplo, el procesador A tiene permiso de escritura sobre X, ha almacenado la información $r(B, X, L)$ y $r(C, X, L)$ y acaba de salir de la barrera L, lo que aconsejará que se envíen copias de escritura de X a B y C especulativamente. Nótese que en este caso a diferencia del caso de locks, no se envía la propiedad de la variable si no solamente una copia de lectura. Los procesadores B y C hacen sus especulaciones correspondientes. Todas las siguientes lecturas que realicen los procesadores serán locales escondiendo así por completo la latencia en cuanto a lectura. El esquema completo se ve en la Figura 5.

La implementación de barreras en PEPE es centralizada y coordinada por el procesador 0. Cada procesador al llegar a una barrera, envía un mensaje al procesador 0 reportando su llegada. Cuando todos los procesadores han llegado, el procesador 0 avisa de esto a todos y pueden continuar con su ejecución. Una forma muy eficiente de implementar la especulación propuesta anteriormente sería entonces incluir en el mensaje que cada procesador manda al procesador coordinador, las variables que se deben enviar especulativamente y a quién deben ser enviadas. Así cuando el procesador 0 envíe el mensaje para que todos continúen su ejecución, podría adjuntar la información enviada por especulación.

Otra posible mejora en la eficiencia sería el especular la invalidación de copias de lectura. Si vemos en la Figura 5 luego de la segunda barrera, cada procesador debe escribir y para esto invalidar todas las variables que estén con permiso de lectura en ese momento. Entonces sería posible también, junto con el mensaje de liberación de la barrera, el enviar mensajes especulativos de invalidación para aprovechar el uso de la red y evitar entonces la posible sobrecarga que produce el gran número de invalidaciones que deben hacerse.

5.3. Análisis de la Heurística

En los casos propuestos anteriormente parece ser que la heurística funciona bastante bien, pero es factible preguntarse que tanto puede llegar a fallar en las predicciones.

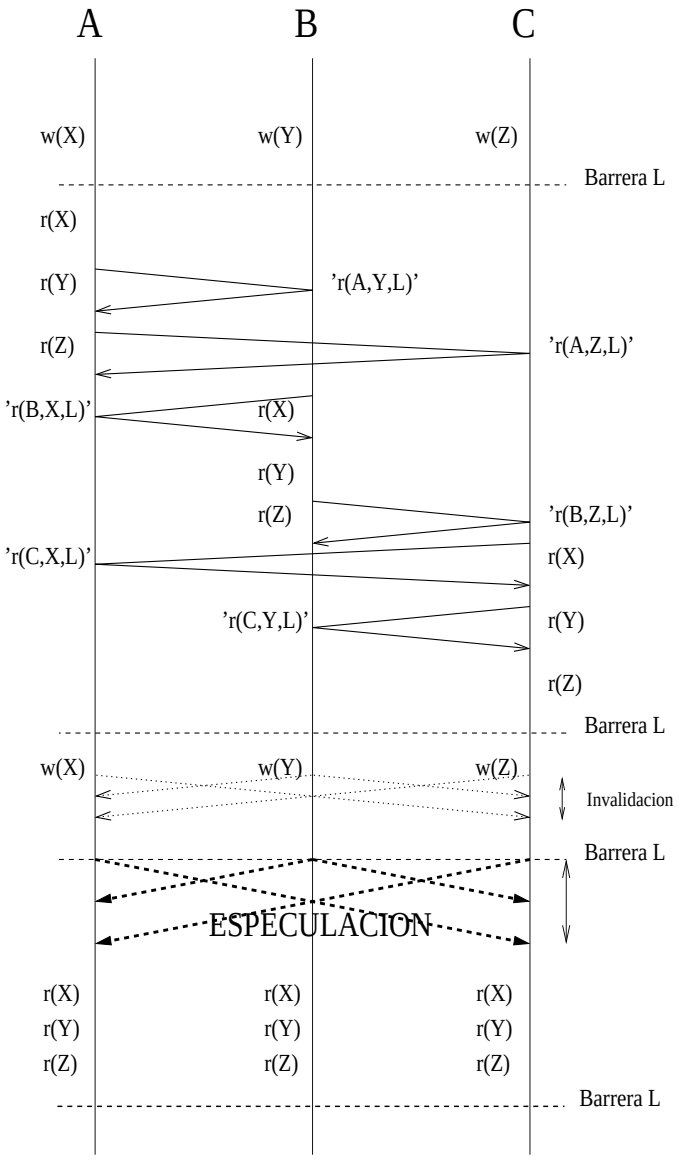

Figura 5: **Especulación para una aplicación con sincronización por barreras. Se especula luego de las escrituras. Las siguientes lecturas son locales.**

Un camino que se debe analizar, es la factibilidad de que el algoritmo pueda darse cuenta de que cometió un error en alguna especulación, y así por ejemplo tomara la decisión de borrar los datos que había almacenado y que lo llevaron a tomar la decisión de especular. Por ejemplo, supongamos que en la Figura 4 en el momento en que A le entrega el lock L (y la propiedad de X) especulativamente a B, y antes de que este último pueda hacer $Aq(L)$ recibe una petición por L, la especulación falló. B entonces podría

enviarle a A un "mensaje sin regreso" en el momento que le pidan L, informándole a A que L le fue solicitado antes de que pudiera usarlo. A debería entonces dejar de enviar L especulativamente a B al hacer $Re(L)$. Más aún, si antes de que B pueda acceder a X le llega una petición por la propiedad de X, B debería informarle a A que también falló en la especulación de la propiedad de la variable, y A debería definitivamente eliminar a $w(B, X, L)$ para no cometer el mismo error. Nótese que si suponemos que el programa está libre de competencia por datos, a B no le puede llegar una petición por la propiedad de X sin que antes le llegue un mensaje $Aq(L)$.

Un algoritmo similar al propuesto en la sección 5.1 se puede hacer para el caso en que tenemos lectores y escritores sincronizados mediante variables de locks. Después de vincular variables compartidas a variable de sincronización, especulativamente después de las operaciones *Re* se enviarán la información actualizada que necesiten otros procesadores. Este método es similar al modelo de consistencia *Release Consistency* (RC) que es un modelo híbrido estrictamente más relajado que el de consistencia secuencial (en RC puede haber comportamientos mucho más extraños que en un modelo de consistencia secuencial) y que usa operaciones de sincronización para mantener la coherencia de memoria. Nuestro método SWDH podría tender a tener una eficiencia cercana al de un protocolo RC para programas correctamente sincronizados. Según lo que nos parece SWDH se ubica en un punto intermedio entre un protocolo de consistencia secuencial (SC) y uno de RC. Si el programador escribe una aplicación sin sincronización alguna, o comete errores en la sincronización, SWDH asegura que la memoria al menos se comportará secuencialmente y tal vez podrá ganar algo en eficiencia. Si el programador escribe una aplicación como para un sistema RC, SWDH puede llegar a eficiencias cercanas y (creemos) que su eficiencia será mejor que la del programa corriendo en un modelo SC común.

6. Conclusiones y trabajos a futuro

En este artículo identificamos las falencias que a nuestro juicio tienen los protocolos especulativos actuales para hardware, y propusimos varias mejoras muchas de las cuales pueden ser implementadas en un sistema de software. Identificamos además algunos de los principales aspectos a considerar en la implementación de un protocolo de este tipo por software.

Se planteó un protocolo especulativo mediante la heurística SWDH aplicable a un sistema DSM en software que implemente consistencia secuencial al estilo de PEPE. SWDH parece ser más eficiente que un modelo de consistencia secuencial común y, a nuestro juicio, puede alcanzar eficiencias cercanas a las de un protocolo de Release Consistency en programas correctamente sincronizados.

Es importante el intentar hacer pruebas de las ideas propuestas en el artículo para poder determinar la factibilidad y la real ganancia de nuestros métodos. El siguiente paso entonces es realizar simulaciones en un monoprocesador para distintos patrones de accesos observados en aplicaciones típicas, y adicionalmente diseñar cargas de trabajo aleatorias que sean exigentes para los algoritmos, para posteriormente implementarlos en un ambiente DSM real. Un posible trabajo a futuro es el intentar extender la heurística SWDH, para poder alcanzar un mayor acierto en las predicciones, así como el identificar de una manera eficiente las malas decisiones que se pudieron tomar en el pasado para no cometerlas nuevamente. Sería interesante también el intentar abarcar otro tipo de modelos de consistencia y poder en ellos aumentar la eficiencia.

Agradecimientos

Los agradecimientos son principalmente para Álvaro Campos, que en paz descanse, quien nos aportó con muchas ideas interesantes en este artículo, principalmente en el análisis de la eficiencia de la heurística propuesta en el sentido de que esta puede funcionar como un protocolo RC en programas correctamente sincronizados. Dedicamos este artículo a él y a su incansable dedicación a la ciencia.

Referencias

[1] D. Koletsou, T. Nightingale, S. Sair, O. Sinanoglu "Performance of Software Distributed Shared Memory Systems".

[2] A. Lai, B. Falsafi, "Memory Sharing Predictor: The Key to a Speculative Coherent DSM", *Proceedings of the 26th Annual International symposium on Computer Architecture*, 1999.

[3] A. Lai, B. Falsafi, "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction", *Proccedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[4] F. Meza, "Memoria Compartida Distribuida en Ambientes de Bajo Costo", *Tesis de Magister Scientiae en Computación*, ITC, Cartago Costa Rica, 2001.

[5] R. Netzer, B. Miller, "Detecting Data Races in Parallel Program Executions", *Workshop on Advances in Languages and Compilers for Parallel Computing*, 1990.

[6] M. Sakr, C. Giles, S. Levitan, B. Hornw, M. Maggini, D. Chiarulli, "On-Line Prediction of Multiprocessor Memory Access Patterns", *Proccedings of the IEEE International Conference of Neural Networks*, p. 1564, 1996.

[7] A. Tanenbaum, "Distributed Operating Systems", *Prentice Hall*, 1995.