Modelos de Consistencia y Migración de Threads en un Sistema de MCD

Alvaro E. Campos y Federico Meza
Departamento de Ciencia de la Computación
P. Universidad Católica de Chile
[acampos,fmeza]@ing.puc.cl

RT-PUC-DCC-99-01 Junio, 1999

Los Reportes Técnicos PUC-DCC pueden solicitarse a:
PUC-DCC Technical Reports can be requested from:

Secretaría de Investigación
Departamento de Ciencia de la Computación (143)
Pontificia Universidad Católica de Chile
Vicuña Mackenna 4860
Santiago 22, Chile

También disponible en la Internet en el URL:
Also available on the Internet under URL:

ftp://ftp.ing.puc.cl/pub/escuela/dcc/techReports/rt99-01.ps

Efectos del Modelo de Consistencia en la Administración de Threads en un Sistema de Memoria Compartida Distribuida

Alvaro E. Campos y Federico Meza
[acampos,fmeza]@ing.puc.cl
Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

Junio, 1999

Resumen

Los sistemas de Memoria Compartida Distribuida (MCD) emulan memoria compartida sobre *hardware* de paso de mensajes, permitiendo construir máquinas paralelas virtuales de amplia escalabilidad, bajo costo y fáciles de programar. El *multithreading* ayuda a mejorar el desempeño de los sistemas de MCD, aprovechando el retardo provocado por los requerimientos remotos. La migración de *threads* permite localizar los accesos a memoria, llevando los procesos hacia los datos. En un esquema de migración de *threads* es necesario estudiar las implicaciones que puede tener el modelo de consistencia empleado, tanto en las acciones que se deben tomar como en el desempeño del sistema resultante. En este trabajo se hace un estudio preliminar al respecto y se propone un proyecto de investigación conducente a la elaboración de una tesis doctoral en el área, apoyada en un amplio componente experimental.

1 Marco Teórico

1.1 Sistemas de Memoria Compartida Distribuida

Los sistemas de Memoria Compartida Distribuida (MCD) [1] son sistemas de *software* que emulan semántica de memoria compartida sobre *hardware* que ofrece soporte sólo para comunicación mediante paso de mensajes. Este modelo permite utilizar una red de estaciones de trabajo de bajo costo como una máquina paralela con grandes capacidades de procesamiento y amplia escalabilidad, siendo a la vez fácil de programar.

Cada uno de los nodos en un sistema de MCD aporta memoria local para construir un espacio global de direcciones virtuales que será empleado por los procesos que se ejecuten en el sistema

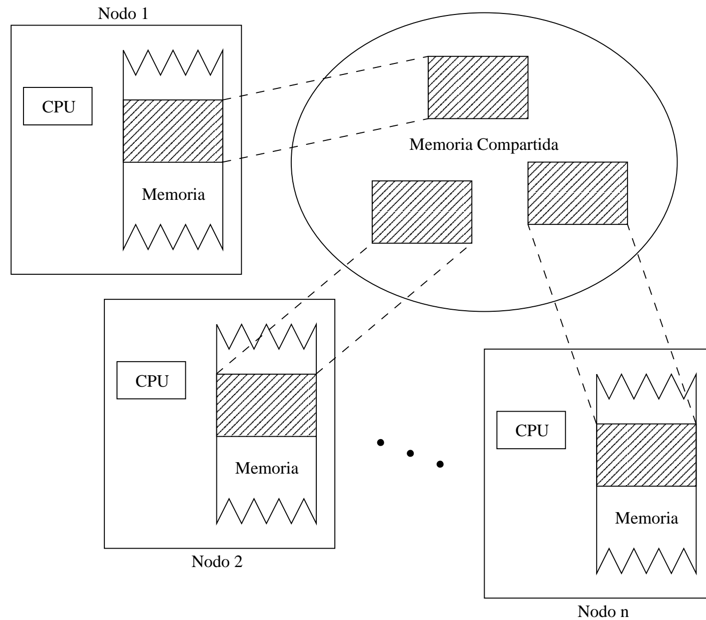

Figura 1: Sistema de Memoria Compartida Distribuida

(Figura 1). El *software* de MCD se encarga de interceptar las referencias a memoria que hacen los procesos y satisfacerlas, ya sea local o remotamente. Si los accesos a memoria hacen referencia a posiciones almacenadas remotamente, es necesario llevar a cabo una transferencia a través de la red con el consecuente *overhead*.

Este estudio se limita a un tipo particular de sistema de *software* de MCD: Basado en páginas, cuya primera implementación fue Ivy [2]. Otras posibilidades son los sistemas basados en variables compartidas (e.g., Munin [3]), y los basados en objetos (e.g., Linda [4]).

En los sistemas basados en páginas la granularidad de la memoria se define a nivel de página de memoria. Esto quiere decir que la mínima unidad referenciable por el sistema de MCD es la página, no pudiendo distinguir entre variables distintas almacenadas en la misma página.

A pesar de que el uso de una granularidad a nivel de página puede introducir problemas como el *false sharing* [5], también permite mantener el *overhead* producido por la administración de la memoria distribuida dentro de parámetros razonables. Esto se debe a que se puede hacer uso de las características de *hardware* para el manejo de memoria virtual, disponibles en la mayoría de los computadores.

En adelante, cuando se mencione los sistemas de MCD se estará haciendo referencia implícita a sistemas basados en páginas.

1.2 Consideraciones de Desempeño y Modelos de Consistencia

El alto tráfico de páginas por la red que puede provocar un sistema de MCD obliga a la utilización de memoria local en cada nodo para almacenar copias de páginas frecuentemente usadas, en la misma forma en que un *caché* ayuda a mejorar el desempeño de un sistema multiprocesador fuertemente acoplado. Sin embargo, la incorporación de estos *cachés* introduce un problema de coherencia para las múltiples copias que pueden existir de cada página.

La actualización de las páginas modificadas puede llevarse a cabo de distintas maneras. En primer lugar, puede emplearse un protocolo de actualización: Se envía las páginas actualizadas a todos los nodos que mantienen copias. Otra opción es simplemente enviar mensajes de invalidación para las copias, de modo que la actualización se lleve a cabo por demanda. Finalmente, puede enviarse *diffs* que codifiquen los cambios a las páginas y que permitan construir la versión actualizada a partir de una página original y múltiples modificaciones. Esta última opción es la que se emplea en los protocolos que soportan múltiples escritores simultáneos sobre una página [6], con el objetivo de reducir el problema de *false sharing*. Sea cual sea el caso aquí se emplea simplemente el término actualización.

Por otra parte, el modelo de consistencia establece la forma en que se hacen visibles a los distintos nodos del sistema las actualizaciones hechas a la memoria compartida. Dado un cierto modelo de consistencia, los programadores y el sistema "acuerdan" un determinado comportamiento para la memoria. Así, bajo un **modelo de consistencia estricto**, cada actualización es vista en forma "instantánea" por todos los nodos en el sistema. Aunque éste sería el ideal para cualquier programador, en la práctica es imposible de implementar y mucho menos en una red de computadores.

Los primeros sistemas de MCD empleaban protocolos rígidos que implementaban un modelo casi estricto, el **modelo de consistencia secuencial, SC** [7]. Bajo este modelo, las actualizaciones a la memoria compartida son vistas en el mismo orden por todos los nodos. Sin embargo, este orden puede variar en distintas ejecuciones, en la misma forma en que podría variar bajo un modelo

estricto dada la concurrencia con que se realizan las actualizaciones. Para poder cumplir con este modelo es necesario hacer del conocimiento de todos los nodos la ocurrencia de una actualización, provocando una alta tasa de comunicación por la red y por consiguiente un desempeño muy pobre del sistema en general.

Con el objetivo de mejorar el rendimiento, disminuyendo la cantidad de mensajes de consistencia que es necesario enviar, se ha propuesto modelos de consistencia más relajados, en el sentido de que son menos rigurosos en la actualización de la memoria compartida. Los programadores deben tener conciencia de esto y programar de acuerdo al modelo establecido. Las restricciones serán mayores conforme se relaje el modelo, pero a la vez se disminuirá la cantidad de comunicación necesaria para mantener la memoria consistente. Cabe resaltar que, por lo general, estas restricciones no representan un gran obstáculo, dadas las técnicas de programación normalmente aplicadas.

El modelo **Release Consistency, RC** [8], se basa en el supuesto de que los accesos a variables compartidas se protejen en secciones críticas empleando primitivas de sincronización, como *locks*. En este caso, todo acceso está precedido por una operación *acquire* y seguido por una operación *release*. Puesto que ningún otro proceso, ni local ni remoto, puede acceder a las variables modificadas protegidas en la sección crítica hasta tener control del *lock*, la actualización de cualquier modificación puede postergarse hasta el momento en que se lleva a cabo la operación *release*.[1] El *release* no se completa hasta que la actualización se haya llevado a cabo en todos los nodos donde haya copias.

El modelo **Lazy Release Consistency, LRC** [9], es una modificación del anterior, que intenta no hacer trabajo innecesario[2], evitando actualizar copias que no serán utilizadas posteriormente. En este caso, la actualización se retarda en cada nodo hasta que algún proceso ejecute un *acquire*, que es el momento en que verdaderamente se requieren en ese nodo los datos actualizados.

El modelo **Scope Consistency, ScC** [10], se basa en la definición de *ámbitos de consistencia*, o simplemente *scopes*. Un *scope* es el entorno respecto al cual se llevan a cabo referencias a memoria. Es decir, únicamente se garantiza que las modificaciones realizadas dentro de un *scope* serán visibles dentro de ese *scope*. Puede visualizarse un *scope* como todas las secciones críticas que se encuentran

[1] De ahí el nombre del modelo: Release Consistency.
[2] Por eso el calificativo lazy, holgazán.

protegidas por un mismo *lock*. El intervalo en el cual un *scope* está abierto en un proceso dado se denomina una *sesión*. Por ejemplo, cuando un proceso se está ejecutando en su sección crítica. Cualquier modificación llevada a cabo dentro de una sesión del *scope* se hace visible a otros procesos que activen sesiones en ese *scope*. No se garantiza que sean visibles las actualizaciones hechas fuera de la sesión.

El modelo **Causal Consistency, CC** [11], determina las actualizaciones que deben propagarse de acuerdo a una *relación de causalidad* entre los accesos a memoria. Dos escrituras a la memoria están causalmente relacionadas si el producto de una de ellas puede depender del producto de la otra, a través de una lectura. De acuerdo a este modelo, las escrituras que están causalmente relacionadas deben verse en el mismo orden en todos los nodos. El resto de las actualizaciones no tienen restricción en su propagación.

Existe gran cantidad de modelos de consistencia que no han sido mencionados aquí [12]. La elección de los modelos descritos se basó en la probada aplicabilidad que han tenido en implementaciones reales de sistemas de MCD. Cabe mencionar Ivy [2] (SC), TreadMarks [13] (LRC), y Brazos [14] (ScC).

1.3 Multithreading y Migración de Threads

En los últimos años los sistemas operativos han evolucionado hacia el soporte de procesos *multithreaded*. Los *threads* son procesos "livianos" que comparten un espacio de direcciones y recursos del sistema operativo [15]. Los *threads* que forman un proceso se diferencian únicamente en los registros y el *stack*. Los cambios de contexto son poco costosos para el caso de *threads* de un mismo proceso, y la cooperación entre ellos es expedita pues comparten memoria sin protección.

En un sistema de MCD el uso de *multithreading* ayuda a mejorar el rendimiento, permitiendo aprovechar el retardo producido por los requerimientos remotos. Cuando un *thread* se bloquea porque debe hacerse una petición por una página ubicada en otro de los nodos del sistema, otro *thread* puede continuar aprovechando el procesador local. A pesar de que este cambio de contexto involucra algún costo, el mayor retardo se da en el tiempo de transmisión y de procesamiento del requerimiento en el nodo remoto [16].

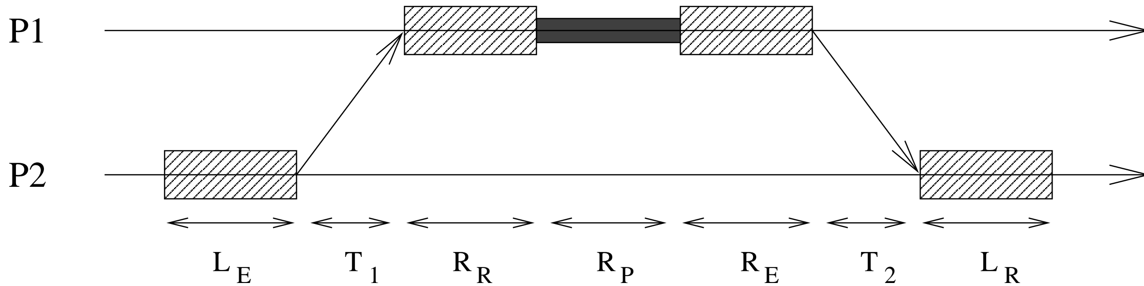

Figura 2: Distribución del Retardo en un Requerimiento Remoto

Este hecho puede apreciarse en la Figura 2. El retardo provocado por un requerimiento remoto puede dividirse en *overhead* local para el envío del requerimiento (L_E), tiempo de transmisión a través de la red (T_1), *overhead* remoto por la recepción del requerimiento (R_R), procesamiento remoto del requerimiento (R_P), *overhead* remoto por el envío de la respuesta (R_E), tiempo de transmisión de la respuesta (T_2) y *overhead* local por la recepción de la respuesta (L_R).

Si bien es cierto que con las tecnologías de red actuales, los tiempos de transferencia son insignificantes comparados con los *overheads* de envío y recepción (L_E, R_R, R_E, y L_R), aún así, si se supone que los *overheads* son iguales en ambos nodos, el tiempo de procesamiento local, $L_E + L_R$, es menos de la mitad del tiempo de retardo total del requerimiento. El resto podría aprovecharse ejecutando otros *threads*.

El uso de *multithreading* también ayuda en la estructuración de las aplicaciones paralelas y en el balanceo de carga entre los procesadores [17]. Además, es posible aprovechar las arquitecturas paralelas simétricas (SMP) ampliamente disponibles en la actualidad. Si el computador cuenta con varios procesadores, todos ellos son aprovechados al existir varios *threads* esperando ejecución.

Por otra parte, puesto que los threads son pequeños y fáciles de administrar, es posible incorporar al sistema una característica de **migración de** *threads*, es decir, poder mover un *thread* que se está ejecutando en un nodo hacia otro nodo del sistema. El objetivo de hacer esto es mejorar la localidad de los accesos a memoria compartida, ubicando en un único nodo aquellos *threads* que acceden a los mismos datos. De esta forma se reduce la cantidad de comunicación entre los nodos y por lo tanto se mejora el desempeño global del sistema [17]. Por supuesto que lo idóneo es que esta migración se haga de forma transparente al programador, y de esta forma los programas no

deban modificarse para adaptarse ante la eventualidad de una migración.

Sin embargo, la tarea de migración no es sencilla. Para migrar un *thread* es necesario suspenderlo en su ubicación actual y reiniciarlo en su nueva ubicación en el mismo estado que estaba anteriormente. El problema es que, si se está empleando un modelo de consistencia relajado, no hay garantía de que la visión de la memoria compartida que tiene luego de la migración sea la misma que tenía en su ubicación original. La situación varía dependiendo del modelo de consistencia que se esté empleando.

Un enfoque que puede ayudar a facilitar la tarea de la migración es mantener todos los datos de un *thread* en la memoria compartida, incluyendo su *stack*. De este modo, cuando el *thread* deba migrar, el sistema de MCD se encarga de gran parte del trabajo de migración en forma transparente. Lo único que deberá migrarse por separado será el estado del procesador (los registros).

El tratamiento de estas problemáticas constituye el punto central de esta investigación y se trata a continuación.

2 Propuesta de Investigación

Cuando un *thread* migra de un nodo a otro, la visión de la memoria compartida que tiene en su nueva ubicación depende del modelo de consistencia que se esté empleando. Por esta razón, es posible que sea necesario tomar acciones de sincronización, implícitas y explícitas, para poder cumplir cabalmente con el modelo establecido.

Se propone llevar a cabo un estudio profundo del efecto que tiene el modelo de consistencia en el funcionamiento de un sistema de MCD con migración de *threads*. Esta investigación tendrá un componente analítico y uno experimental. Se pretende caracterizar las situaciones en que el comportamiento del sistema varía dependiendo del modelo de consistencia.

El caso más simple es el del modelo de consistencia secuencial. Bajo este modelo se han implementado sistemas de migración, como es el caso de Millipede [17], dado que no presenta problemas mayores. Bajo SC, las actualizaciones a la memoria se propagan sin retardos, respecto al proceso de migración, por lo que un *thread* que migra a otro nodo encuentra la misma visión de

la memoria que tenía en su ubicación original. En otras palabras, el *thread* llega a un nodo que ha visto las actualizaciones a la memoria en el mismo orden en que fueron vistas en el nodo origen del que proviene. Sin embargo, los casos en que se está empleando un modelo de consistencia más relajado que el secuencial requieren de un estudio más cuidadoso.

2.1 Marco General para la Migración

Es necesario definir un marco general para el estudio de la migración, de modo que pueda llevarse a cabo comparaciones y, a la vez, se evite algunos problemas que podrían producirse en situaciones extremas. Existen algunas restricciones sobre el proceso de migración [17] que deben ser tomadas en cuenta.

En primera instancia, se supone que las migraciones nunca ocurren cuando el proceso a migrar se encuentra dentro de una sección crítica. Esta restricción es lógica y no resulta muy limitante, si se piensa que durante una sección crítica el proceso se encuentra en poder de recursos locales que no pueden ser migrados. La generalidad que se puede obtener no justifica el costo de permitir la migración bajo estas circunstancias. Más adelante se analiza la posibilidad de obviar esta restricción.

Una restricción similar se aplica cuando un *thread* tiene asignados recursos del sistema que se administran localmente. Este es el caso de impresoras, archivos locales, etc. Bajo estas circunstancias, al *thread* no le es permitido migrar.

Además, la interacción entre las acciones de consistencia que lleva a cabo el sistema de MCD y los procesos de migración no deben interferir entre sí. Para esto, se restringe la migración de *threads* para que ocurra únicamente cuando no existan acciones de consistencia en ejecución o pendientes de finalización.

Por otra parte, para llevar a cabo el estudio se supone que el *thread* migrado llega a un nodo en el que se encuentran copias de las páginas que éste utiliza. En caso contrario el análisis resulta trivial, pues las páginas se obtienen por demanda desde una fuente consistente, en concordancia con el protocolo empleado. Cabe resaltar que la migración de un *thread* puede provocar que el sistema de MCD se vea forzado a transferir páginas que de otro modo no hubiera transferido.

Finalmente, se supone que las modificaciones a datos compartidos se llevan a cabo siempre dentro de secciones críticas. Esta condición es incluso necesaria para protocolos como RC, LRC y ScC.

Es necesario observar que el alcance de esta investigación no abarca las políticas de migración de los *threads*. Este tema constituye, por sí mismo, un problema independiente. Aquí se supondrá que los *threads* son migrados de acuerdo con algún criterio y se limitará el estudio al efecto del modelo de consistencia en el resultado de esta migración.

2.2 Migración bajo Consistencia Secuencial

Como se mencionó anteriormente, cuando se emplea SC la migración de *threads* no presenta mayor problema desde el punto de vista de la consistencia. Con SC, las actualizaciones a la memoria se propagan en forma atómica respecto a la migración, a todos los nodos que mantienen copias locales de las páginas modificadas.

Cuando un *thread* es migrado, en el nodo destino se encuentran copias secuencialmente consistentes de las páginas que emplea, por lo que no es necesario tomar acciones adicionales de consistencia.

2.3 Migración bajo Release Consistency

Cuando el sistema se encuentra bajo RC, la suposición de que no se produce migración dentro de secciones críticas garantiza la consistencia de la memoria en el nodo destino.

En RC, la propagación de las actualizaciones se lleva a cabo al momento del *release*, es decir, al finalizar la sección crítica. El *release* no se da por terminado hasta que la memoria se encuentra consistente en todos los nodos que mantienen copias de las páginas modificadas. Por lo tanto, la migración no es posible hasta que se haya llevado a cabo esta sincronización.

Cuando la migración se lleva a cabo, el *thread* migrado encuentra la memoria en un estado consistente, de acuerdo al modelo RC.

Consideremos el ejemplo de la Figura 3, en el cual un proceso modifica una determinada página en dos ocasiones. Si el *thread* migra en algún punto entre ambas secciones críticas, encuentra la

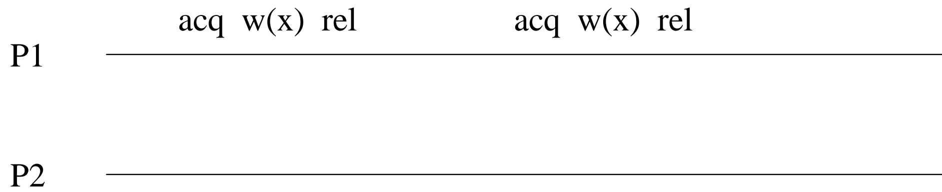

Figura 3: Ejecución de un Thread en un Sistema con 2 Procesadores

memoria consistente en el nodo destino, pues ésta se actualiza al finalizar la primera sección crítica, como se muestra en la Figura 4.

2.4 Migración bajo Lazy Release Consistency

El caso de LRC requiere mayor estudio. No es posible aplicar el mismo razonamiento que para el caso de RC, pues las modificaciones no se llevan a cabo al terminar la sección crítica, sino justo antes de empezar la siguiente.

El aspecto importante en este caso es la obligatoriedad de emplear secciones críticas para modificar datos compartidos. Cuando un *thread* recién migrado intenta acceder a datos compartidos debe primero adquirir un *lock* para entrar a la sección crítica, y en ese momento se lleva a cabo la propagación de las modificaciones hechas en otros procesadores, entre los que puede encontrarse el nodo origen del *thread*, si es del caso. Cumpliendo con el protocolo, el *acquire* no se da por terminado hasta que esta propagación ha concluido.

Volviendo al ejemplo de la Figura 3. Si el *thread* migra entre ambas secciones críticas se produciría una situación como la de la Figura 5. Antes de iniciar la segunda sección crítica el modelo de consistencia actualiza la memoria en el nodo destino para que el *thread* migrado tenga la misma visión que tenía en el nodo origen. Obsérvese que en este caso el sistema de MCD toma acciones

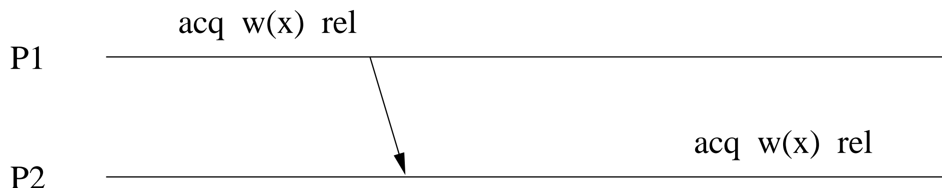

Figura 4: Acciones de Consistencia al Migrar un Thread bajo RC

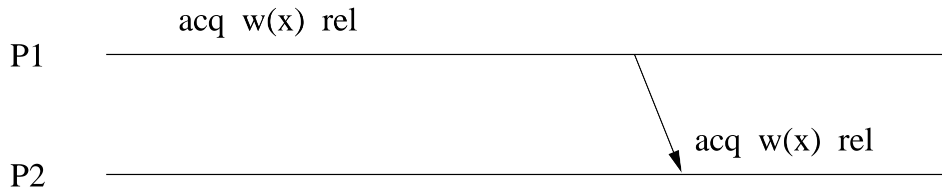

Figura 5: Acciones de Consistencia al Migrar un Thread bajo LRC

de consistencia adicionales como consecuencia de la migración del *thread*.

2.5 Migración bajo Scope Consistency

El caso de ScC es muy similar al de RC y LRC. Los cambios llevados a cabo en una sesión de un *scope* serán propagados a aquellos nodos en que se inicien sesiones del mismo *scope*.

Dada la restricción planteada sobre la no interferencia de las acciones de consistencia y las de migración, y según la definición de ScC, un *thread* no podrá migrar mientras tenga sesiones abiertas de cualquiera de sus *scopes*.

Supongamos que un *thread* logra migrar. Cuando intente acceder a datos compartidos en su nueva ubicación, deberá hacerlo dentro de una sesión de un *scope* particular. De acuerdo a las reglas de consistencia de ScC, antes de permitirle continuar con la sesión, se harán todas las actualizaciones que no se encuentren propagadas, incluyendo aquellas que provengan del nodo origen del *thread* migrado.

Como se ve, también en este caso el protocolo de consistencia se encarga de hacer que el *thread* encuentre la memoria en un estado coherente con la visión que tenía en su ubicación original, en concordancia con el modelo establecido. Algunas de estas acciones puede que hayan sido originadas a raíz de la migración.

3 Trabajo Existente

Los tópicos expuestos en este trabajo han sido tratados únicamente en forma independiente. El tema del *multithreading* en sistemas de MCD se introdujo por primera vez en TreadMarks [16] y más recientemente se ha incorporado en sistemas como Millipede [18] y Brazos [14].

A su vez, el tema de la migración de *threads* se ha tratado desde hace más de una década, pero no en sistemas de MCD. Puede citarse como un buen ejemplo el sistema Charlotte [19].

El único trabajo en que se encuentra concordancia con los objetivos aquí expuestos es el realizado en Millipede [17], un sistema de MCD de reciente aparición. Sin embargo, la característica de migración de *threads* en Millipede se restringe al uso de un modelo de consistencia secuencial, lo que resulta bastante limitante.

4 Hipótesis de Trabajo

Se ha establecido que el uso de *multithreading* ayuda a aprovechar productivamente gran parte del retardo provocado por los requerimientos remotos de páginas. A su vez, la migración de *threads* intenta reducir el flujo de páginas entre los nodos del sistema, llevando los *threads* hacia las páginas que están utilizando.

Sin embargo, la obtención de estos beneficios podría depender de una serie de aspectos, como el modelo de consistencia empleado, los criterios para la migración de *threads* y las características de la aplicación que se esté ejecutando.

La migración de un *thread* provoca acciones de consistencia adicionales, que varían dependiendo del modelo empleado. El efecto pareciera ser menor en los modelos estrictos y mayor en los más relajados.

El proceso de migración de *threads* debe someterse a una serie de restricciones que aseguran su correcto funcionamiento. Algunas de estas restricciones son difíciles de evadir, como es el caso de las situaciones en que un *thread* posee recursos del sistema administrados localmente en cada nodo. Sin embargo, resulta interesante estudiar la posibilidad de relajar las restricciones relacionadas a las secciones críticas y a la atomicidad entre las migraciones y las acciones de consistencia. Es probable que los resultados obtenidos también dependan del modelo de consistencia aplicado.

El objetivo es sacar el mayor provecho posible del uso de *multithreading* y de migración de *threads* y, si es del caso, caracterizar aquellas situaciones en las que es mejor no hacer uso de ellos.

5 Plan de Trabajo

La investigación propuesta tiene un componente analítico y uno experimental, siendo ambos de vital importancia para alcanzar los objetivos propuestos.

El análisis llevado a cabo sobre el efecto del modelo de consistencia en la migración es un tanto informal. Es necesario plantear las distintas situaciones siguiendo un formalismo más estricto, que permita llegar a conclusiones certeras para cada caso.

La forma más exacta para lograr esto es tratando directamente las implementaciones en *software* de los protocolos de consistencia. Por su importancia práctica este estudio se basará en tres modelos particulares. LRC, por su probado buen desempeño y disponibilidad de implementaciones de dominio público, como CVM [20] y Quarks [21]. ScC, por su auge en sistemas de MCD emergentes, como Brazos [14]. CC, por su independencia teórica con los demás modelos y por contar con un protocolo construido localmente [22] que se encuentra en experimentación y ha mostrado buen desempeño.

Por otra parte, para analizar el efecto que tiene el uso de *multithreading* y de migración de *threads* sobre el desempeño del sistema de MCD, será necesario llevar a cabo una gran cantidad de experimentación.

En esta etapa se pretende emplear un sistema de MCD actualmente en desarrollo para computadores personales [23]. El sistema permitirá, en una plataforma de muy bajo costo, ejecutar aplicaciones paralelas sobre memoria compartida distribuida. Variando parámetros como el modelo de consistencia empleado y la cantidad de *threads* por aplicación, se obtendrá información estadística sobre el comportamiento del desempeño ante la introducción del *multithreading*, para cada protocolo de consistencia en estudio. Con esta información se podrá validar los resultados teóricos obtenidos. En la siguiente sección se expone el diseño experimental que se empleará.

6 Diseño Experimental

El componente experimental de esta investigación es tan importante como el analítico. Se propone llevar a cabo una serie de comparaciones que permitan caracterizar las situaciones en las que resulta

más conveniente el uso de migración de *threads*.

Puesto que el principal parámetro a considerar es el modelo de consistencia, es necesario contar con una implementación apropiada para cada protocolo considerado. Además, puede ser necesario manipular los programas para adaptarse a las características de cada modelo en particular.

Sin embargo, para mantener la justicia y por tanto la validez de las comparaciones, se debe tratar de utilizar la mayor cantidad de código común, tanto en los programas de prueba como en los mecanismos de comunicación y sincronización.

La elección de las aplicaciones también es un aspecto delicado. Los programas no deben incluir características que introduzcan complicaciones o particularidades adicionales a las que acompañan al modelo de consistencia en uso.

Se planea emplear un conjunto de programas tradicionales, cuidadosamente seleccionados de los *suites* de aplicaciones *SPLASH* [24], *SPLASH-2* [25] y *NAS* [26].

En la mayoría de los experimentos se empleará como grupo de control un sistema *multithreaded* pero sin migración. Los criterios que servirán para llevar a cabo las comparaciones son la mejora de desempeño o *speedup*, la cantidad de mensajes de consistencia que transitan por la red, el número de *page misses* que se producen, e incluso la carga de trabajo de cada nodo.

Como plataforma de experimentación se empleará un sistema de MCD actualmente en desarrollo [23] sobre un grupo de aproximadamente 20 computadores Pentium con Windows NT. La red es una Ethernet de 10 Mbps.

Dependiendo del financiamiento disponible se pretende ampliar la experimentación a una red más rápida basada en ATM, así como a una pequeña red de computadores con varios procesadores en una arquitectura simétrica (SMP). Esta última alternativa requiere de un estudio cuidadoso pues el uso de varios procesadores puede introducir un problema de contención en la interfaz de red que entorpecería la experimentación.

Referencias

[1] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[2] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, 1986.

[3] J.K. Bennett, J.K. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 168–176, 1990.

[4] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[5] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, 1997.

[6] Peter J. Keleher. The Relative Importance of Concurrent Writers and Weak Consistency Models. In *16th International Conference on Distributed Computing Systems*, 1996.

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

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *17th International Symposium on Computer Architecture*, pages 15–26, 1990.

[9] Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *19th Annual International Simposium on Computer Architecture*, 1992.

[10] L. Iftode, J. P. Singh, and Kai Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[11] P.W. Hutto and M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. In *10th IEEE International Conference on Distributed Computing Systems*, pages 302–311, 1990.

[12] David Mosberger. Memory Consistency Models. Technical report, Department of Computer Science, University of Arizona, 1993.

[13] Peter Keleher. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[14] Evan Speight and John K. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the First USENIX Windows/NT Workshop*, 1997.

[15] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, USA, 1995.

[16] Kritchalach Thitikamol and Peter Keleher. Per-Node Multithreading and Remote Latency. *IEEE Transactions on Computers*, 47(4):414–426, 1998.

[17] Ayal Itzkovitz, Assaf Schuster, and Lea Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *Journal of Systems and Software*, 42(1):71–87, 1998.

[18] Roy Friedman, Maxim Goldin, Ayal Itzkovitz, and Assaf Schuster. Millipede: Easy Parallel Programming in Available Distributed Environments. *Software: Practice and Experience*, 27(8):929–965, 1997.

[19] Yeshayahu Artsy and Raphael Finkel. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, 22(9):47–56, 1989.

[20] Peter Keleher. CVM: The Coherent Virtual Machine. Technical report, Department of Computer Science, University of Maryland, 1997.

[21] Dilip Khandekar. Quarks: Portable Distributed Shared Memory on Unix. Technical report, Computer Systems Laboratory, Department of Computer Science, University of Utah, 1996.

[22] Juan E. Navarro. Un Protocolo de Consistencia Causal para Memoria Compartida Distribuida. Master's thesis, Departamento de Ciencia de la Computación, Pontificia Universidad Católica de Chile, Santiago, Chile, 1996.

[23] Federico Meza. Memoria Compartida Distribuida en Ambientes de Bajo Costo. Master's thesis, Centro de Investigaciones en Computación, Instituto Tecnológico de Costa Rica, 1999. Actualmente en desarrollo.

[24] J.P. Singh, W.D. Weber, and A. Gupta. SPLASH: Standford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, 1991.

[25] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *23rd Annual Symposium on Computer Architecture*, 1995.

[26] D. Bailey. The NAS Parallel Benchmark. Technical Report TR RNR-91-002, NASA Ames, 1991.