Migración de *Threads* en un Sistema de Memoria Compartida Distribuida sobre un Multicomputador Homogéneo

Jesús Figueroa, Alvaro E. Campos, Cristian Ruz
[jefiguer,acampos,cruz]@ing.puc.cl
Depto. de Ciencia de la Computación
P. Universidad Católica de Chile
Casilla 306 - Santiago 22 - CHILE
Fax +56-2-354-4444

Federico Meza
fmeza@utalca.cl
Depto. de Ciencias de la Computación
Universidad de Talca
C. Los Niches Km. 1 - Curicó - CHILE
Fax +56-75-325-958

Resumen

Los sistemas de memoria compartida distribuida ofrecen al programador un espacio virtual de memoria compartida sobre un multicomputador con memoria distribuida. De esta forma, es posible construir sistemas paralelos escalables y de bajo costo, pero a la vez, fáciles de programar. El *multithreading* ayuda a mejorar el desempeño de estos sistemas al permitir el traslape de la comunicación con la computación. A su vez, la migración de *threads* ayuda a balancear la carga entre los procesadores pero sobre todo aumenta la localidad de referencia, manteniendo juntos los datos y los *threads* que los usan. En este artículo presentamos la implementación de un mecanismo de migración de *threads* para el sistema de memoria compartida distribuida `DSM-PEPE`, utilizando una biblioteca de *threads* a nivel de usuario. Nuestra implementación resuelve de forma simple los problemas inherentes al proceso de migración sobre un multicomputador homogéneo.

Palabras clave. Migración de *threads*, migración de procesamiento, *multithreading*, memoria compartida distribuida, sistemas distribuidos, multicomputadores.

1. Introducción

El uso de multicomputadores con memoria distribuida representa una alternativa conveniente frente a multiprocesadores de memoria compartida, como herramientas para el procesamiento de alto desempeño. Aunque en los multiprocesadores la semántica que se usa para escribir programas es simple, son costosos y poco escalables, mientras que los multicomputadores utilizan *hardware* convencional y son fácilmente escalables.

Por otra parte, la programación de un multicomputador es más complicada, pues debe controlarse explícitamente el flujo de datos entre las memorias distribuidas. En este sentido, los sistemas de memoria compartida distribuida tienen cabida como una manera de implementar multicomputadores escalables para procesamiento en paralelo de aplicaciones que requieren gran cantidad de cómputos, y que a la vez sean fáciles de programar.

Los sistemas de memoria compartida distribuida (MCD) proveen un espacio virtual de memoria compartida sobre un multicomputador de memoria distribuida [1], permitiendo a los distintos procesos, físicamente distribuidos, compartir una memoria formada a partir de trozos que cada proceso aporta.

Cuando un proceso requiere datos de la memoria compartida que no se encuentran en su memoria local, el sistema de MCD, de acuerdo al protocolo de consistencia de memoria utilizado, hace que los datos sean traídos en forma transparente, para que puedan ser accedidos en forma local. De esta manera el programador puede escribir programas utilizando una semántica de memoria compartida y no debe preocuparse de la comunicación entre las memorias distribuidas. Nuestro trabajo se concentra en sistemas de MCD implementados completamente por *software* y a nivel de aplicación, utilizando *hardware* convencional, y sin participación especial de parte del compilador.

Los principales problemas de los sistemas de MCD son el costo de comunicación por la red y el *overhead* que introduce la administración de la MCD. El primero de estos problemas es inherente a los multicomputadores. En el segundo caso se han estudiado distintas técnicas para reducir el intercambio de datos entre los procesadores. Esto incluye el desarrollo de protocolos de consistencia de memoria relajados [2, 3, 4, 5], el uso de *multithreading* [6] y la migración de *threads* [7, 8].

En general, el uso de *multithreading* es útil pues mejora la estructura de los programas concurrentes, subdividiendo el procesamiento en varias subtareas [9]. Además, la creación, los cambios de contexto y la administración de los *threads* son simples y poco costosos. En un sistema de MCD, el *multithreading* a nivel de aplicación ayuda a reducir los retardos producidos por los requerimientos remotos, ayudando a traslapar computación con comunicación [6].

Por su parte, la migración de procesamiento pretende distribuir la carga de trabajo de manera uniforme, de modo que no existan procesadores sobrecargados, ni tampoco procesadores ociosos mientras exista trabajo pendiente [10, 11]. En un ambiente de *multithreading*, la alternativa de migrar *threads* resulta natural y eficiente [12] ya que la cantidad de información que define al flujo de procesamiento, en el caso del *thread*, es menor que en el caso de un proceso.

En un sistema de MCD, la migración de *threads* permite aumentar la localidad de referencia de los datos, llevando los *threads* en forma dinámica hacia los datos que utilizan [7]. Así, no sólo los datos viajan entre los procesadores conforme son utilizados, sino también se intenta agrupar en un mismo proceso aquellos *threads* que utilizan frecuentemente los datos ahí residentes.

Evidentemente, existe un compromiso entre el balance de carga y el aumento en la localidad de referencia. Mientras el primero intenta maximizar el aprovechamiento de los procesadores, el segundo trata de minimizar la comunicación entre ellos. La política de migración del sistema debe lograr un balance adecuado entre estos dos aspectos.

La migración de *threads* se puede ver como una migración de procesamiento en la que un hilo de control dentro del programa es suspendido en algún punto de su ejecución, transportado a otro proceso y reactivado, continuando su ejecución desde el punto en que fue suspendido en el proceso original. Lo que se migra de un proceso a otro es todo aquello que define al *thread* en el estado en que quedó al momento de ser suspendido y que permite su reactivación.

Un aspecto que se debe considerar es la potencial heterogeneidad del sistema multicomputador, pues desde el punto de vista de la migración, los desafíos y problemáticas son distintos. A diferencia de un sistema en que los computadores tienen iguales características de *hardware* y *software*, en los sistemas heterogéneos es necesario tomar en cuenta que la manera en que se representan internamente los datos e instrucciones en la memoria, podrían diferir entre sistemas que presentan *hardware* o *software* distintos.

Por otra parte, los *threads* pueden ser implementados a nivel de sistema operativo o a nivel de usuario [9]. Los *threads* a nivel de sistema operativo hacen participar al núcleo en su creación y cambios de contexto, lo que encarece su uso en términos de rendimiento global del sistema. Además, su dependencia del sistema operativo los hace poco portables. Por el contrario, los *threads* a nivel de usuario son más portables, y su administración y cambios de contexto son menos costosos. Sin embargo, por el desconocimiento que el sistema operativo tiene de los *threads*, el bloqueo de un *thread* provoca el bloqueo de todo el proceso.

En este artículo se presenta la implementación de un mecanismo de migración de *threads* para el sistema de memoria compartida distribuida `DSM-PEPE` [13, 14]. El sistema opera sobre un multicomputador homogéneo y utilizando una biblioteca de *threads* a nivel de usuario [15].

El artículo se distribuye de la siguiente manera: inicialmente presentamos los conceptos generales de la migración de *threads* y los problemas involucrados; a continuación detallamos la implementación de nuestro mecanismo de migración, incluyendo las características de la biblioteca de *threads* y del sistema de MCD que utilizamos, así como nuestra solución para algunos de los problemas inherentes al proceso; posteriormente enumeramos algunos trabajos relacionados y, finalmente presentamos nuestras conclusiones y el trabajo que estamos llevando a cabo actualmente.

2. Generalidades acerca de la migración

La migración de *threads* consiste en suspender a un *thread* en cierto estado de su ejecución y reactivarlo en otro proceso, en el mismo estado en que fue suspendido. Para lograr esto es necesario rescatar la información que define el estado del *thread* –llámese *stack* y registros–, transportar esa información al otro procesador, para luego, de alguna manera, cargarla en otro *thread* que asumirá la ejecución en el estado en que el *thread* original fue suspendido.

En el caso más general, con la migración de procesamiento se busca distribuir dinámicamente la carga de trabajo entre los procesadores que conforman el sistema y así poder asegurar que ninguno de ellos se encontrará desocupado o con poco trabajo, mientras otros estén sobrecargados con procesamiento [10, 11]. Cuando se está resolviendo problemas con comportamiento no uniforme o cuando el sistema en el que se trabaja varía en cuanto a su composición, es necesario que el balance de carga sea dinámico. La distribución estática presenta varias desventajas, no solamente por no aplicarse a las situaciones recién descritas, sino porque frente a distribuciones iniciales erróneas o mal elegidas, puede generar un gran *overhead* por comunicación.

Por otra parte, cuando se utilizan redes de computadores personales para el procesamiento, los usuarios de esos computadores deben poder utilizar todos los recursos disponibles. Luego, si un usuario llega a una estación de trabajo que se encuentra procesando, los *threads* ejecutando en ella tienen que ser llevados a otros nodos para continuar con su procesamiento y así no degradar la respuesta que percibe el usuario.

Por otro lado, la migración de *threads* permite aumentar la localidad de referencia de los datos [7], ya que los *threads* que los utilizan son llevados a donde éstos se encuentran. De esta manera, se pueden agrupar los *threads* y los datos que éstos ocupan con mayor frecuencia en el mismo procesador. Es claro que la comunicación debido al acceso a los datos compartidos puede disminuir, ya que las referencias serán en su mayoría locales.

2.1. Migración del *stack*

Una de las principales dificultades en el proceso de migrar un *thread* es el manejo del *stack* y su coherencia respecto a los datos que ahí se encuentran. Dentro del *stack* puede haber variables locales, registros de activación y, en general, direcciones de memoria que pueden apuntar a posiciones dentro o fuera del *stack*.

En el caso de las direcciones de memoria que apuntan al *stack*, si éste es colocado en el proceso destino en una ubicación distinta a la original, las direcciones estarán desfasadas por lo que serán inválidas. Las direcciones de memoria que hacen referencia a posiciones del *heap*, que no están en la MCD, también representan un problema pues los datos no estarán en el proceso destino, dado que no fueron migrados junto con el *stack*. Por otro lado, las direcciones de memoria que apuntan fuera del *stack* pero a la memoria compartida distribuida son siempre válidas pues se trata de direcciones en el espacio global.

Estos problemas no se presentan cuando se restringe a los procesos a guardar sus datos en el espacio global de direcciones o en variables locales, pero no en el *heap*, y se garantiza además que el *stack* mantendrá su ubicación después de cualquier migración. Este es el enfoque que seguimos en nuestro trabajo.

3. Implementación del mecanismo de migración

Nuestro mecanismo de migración de *threads* fue construido sobre un sistema de memoria compartida distribuida cuyos programas de usuario incorporan *multithreading* a través de una biblioteca. El sistema incorpora todas las facilidades para la implementación de la migración y de la política que la lleva a cabo. Este trabajo se concentra en el mecanismo de migración y deja de lado la política, la cual puede ser incorporada en cualquier momento como un módulo independiente. Básicamente, la política decide cuándo actúa nuestro mecanismo, sobre cuáles *threads*, y sobre cuáles procesos. Para efectuar la migración de un *thread*, el módulo que implementa la política de migración debe invocar a la función `migrar()`, indicando el identificador del *thread* que se quiere migrar y el nodo destino de la migración.

3.1. Biblioteca de *threads*

La biblioteca que se ha utilizado implementa *threads* a nivel de usuario, evitando los problemas de bloqueo que tienen implementaciones similares. En nuestra biblioteca, cuando se bloquea a un *thread* no se detiene a todo el proceso. Además, la biblioteca es portable a otros sistemas operativos y arquitecturas de *hardware*, siendo necesarias sólo algunas modificaciones en el manejo de los cambios de contexto, cuya implementación es dependiente de la manera en que se manejan las instrucciones y registros dentro de cada arquitectura.

Para la representación de un *thread* se utiliza una estructura que incluye campos para almacenar registros –como el *stack pointer*–, información del *thread* –como su identificador dentro de la biblioteca–, y el puntero a la función que ejecuta el *thread*, entre otros. Todos estos campos se almacenan al principio de la estructura que representa al *thread*. A continuación se encuentra el *stack* del *thread*, que crece de manera inversa, y el cual tiene un tamaño predeterminado. Por último, al final se disponen los argumentos que recibe la función del *thread*. Esta organización se muestra en la Figura 1.

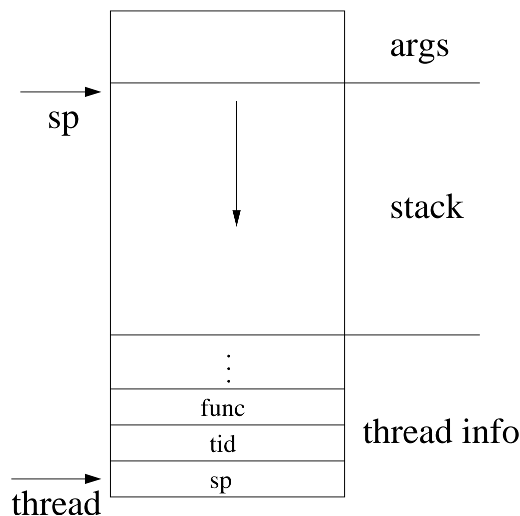

Figura 1: Representación de un *thread*: argumentos, *stack* e información del *thread*

La biblioteca fue modificada y se agregaron dos funciones que tienen por objetivo entregar más información útil respecto al *stack*. La primera de ellas –función `tgetstack()`– retorna la dirección de memoria donde comienza la estructura que contiene el *stack*. La segunda –función `tgetstacksize()`– retorna el tamaño en bytes de la estructura que representa al *thread*. También se incluyó una cola adicional, llamada `SUSPENDED`, similar a la que se utiliza para bloquear en fila a los *threads* que esperan en semáforos. En esta cola se bloquean los *threads* que pueden ser migrados o aquellos *threads* que esperan que otro migre sobre ellos, en la forma que explicamos más adelante. La cola `SUSPENDED` es manejada por las funciones `tsuspend()` –que coloca a un *thread* en la cola– y `tresume()` –que saca el *thread* de la primera posición de la cola.

La estructura que representa al *thread* contiene el *stack*, los registros y el resto de la información relacionada con el *thread*, es decir, el estado en que el *thread* fue suspendido. Entonces, esta estructura es precisamente lo que se debe migrar.

3.2. `DSM-PEPE`

`DSM-PEPE` es un sistema de memoria compartida distribuida implementado por *software*, y que opera sobre multicomputadores compuestos por computadores personales con Windows o Linux [13]. Está escrito en C++ y utiliza paso de mensajes como mecanismo de comunicación; sin embargo se oculta esta implementación al programador, a través de una arquitectura basada en capas y módulos. Su diseño facilita la portabilidad a distintos sistemas operativos, separando los componentes dependientes del *hardware* o del sistema operativo, del resto del sistema.

El diseño modular permite separar aspectos como el paso de mensajes entre nodos, implementado a través de *sockets*, utilizando una abstracción de alto nivel –la clase de objetos `PostOffice`. Otros módulos se encargan de manejar las acciones de sincronización entre nodos, permitiendo utilizar barreras y *locks*. También se utilizan objetos que encapsulan la representación de la memoria compartida.

La memoria compartida distribuida está disponible en `DSM-PEPE` a través de regiones de memoria dentro de las cuales se crean las variables que van a ser compartidas entre los distintos procesos distribuidos. El sistema implementa dos protocolos de consistencia de memoria: consistencia secuencial y consistencia de entrada, permitiendo utilizar regiones de memoria distintas bajo distintos protocolos de consistencia.

Los programas de usuario siguen el esquema SPMD (*Single Program Multiple Data*), es decir, un único programa que se ejecuta en todos los procesadores, y que actúa en cada uno de ellos sobre distintos conjuntos de datos. Así, cuando se ejecuta un programa en `DSM-PEPE`, se utiliza el mismo código en todos los procesos, aunque pueden operar en forma diferenciada a partir de su identificador de proceso.

`DSM-PEPE` posee *multithreading* al nivel del programa de usuario, y es en este contexto en el que se desarrolla nuestro sistema de migración de *threads*. Puesto que actualmente no se ha implementado una política de migración, las migraciones son dirigidas por los mismos *threads* con fines experimentales.

Durante el proceso de una migración, toda la comunicación entre procesos involucrada se lleva a cabo utilizando el módulo de comunicación de `DSM-PEPE`. De esta forma se garantiza la portabilidad a diversas plataformas y se logra una mejor integración con el sistema de MCD.

3.3. Solución al problema del *stack*

Se decidió trabajar sobre un ambiente homogéneo de manera de no lidiar, por ahora, con el problema de representación de datos, direcciones e instrucciones entre una plataforma y otra. Sin embargo, nuestro mecanismo funciona en las distintas plataformas en que opera `DSM-PEPE`. Por otra parte, para garantizar portabilidad, nuestra implementación no utiliza ningún tipo de preprocesamiento, como el que se usa en MigThread [16], sino que toda acción de migración se lleva a cabo en tiempo de ejecución, como en Millipede [8].

Para evitar problemas con las direcciones del *stack*, nuestra implementación replica los *threads* en todos los procesos. Es decir, por cada *thread* creado en un proceso, se crea otro en cada uno de los demás procesos del sistema. De esta forma, cada *thread* en un proceso tiene una imagen o *thread* equivalente en todos los demás procesos, los que se encuentran bloqueados en espera de una migración. En este esquema, cuando se desea migrar un *thread* de un proceso a otro, se utiliza el *thread* imagen en el nodo destino, simplemente copiando la estructura del *thread*, en la posición de memoria correspondiente. Por ejemplo, si se tienen N nodos y en cada uno se utilizan M *threads* distintos, entonces en la práctica se crean $(N \times M) \times N = N^2 \times M$ *threads*.

A pesar de que la cantidad de *threads* que se utilizan puede ser alta, su costo es muy bajo –aproximadamente 1 KB– y la biblioteca de *threads* no impone limitaciones en cuanto al número máximo de ellos que se puede tener activos en forma simultánea.

Nuestra propuesta para atacar el problema de las direcciones del *stack* es que los *threads* equivalentes ubiquen sus estructuras, y por ende su *stack*, en las mismas posiciones de memoria en todos los procesos. De esta manera, sin importar el proceso en que se encuentre, el contenido del *stack* se mantendrá coherente.

Lo anterior, sumado a que el sistema es homogéneo, y al hecho de que el programa que ejecuta cada proceso es el mismo, nos permite asegurar que en cada proceso la asignación de memoria local será la misma.

Se puede apreciar que el problema de las indirecciones que existan dentro del *stack* hacia sí mismo no lo es en este esquema, ya que, como son las mismas direcciones de memoria, el desfase no existe. Por otro lado, el problema de las direcciones de memoria fuera del *stack* también se soluciona en parte, ya que si un puntero referencia a una dirección de memoria fuera del *stack*, pero dentro del espacio de memoria compartida, la consistencia de la memoria es garantizada por `DSM-PEPE`. El caso en que un puntero dentro del *stack* referencie a una dirección de memoria local fuera del espacio compartido, no puede presentarse por las restricciones impuestas.

3.4. Detalles de la implementación

Inicialmente el sistema DSM crea un *pool* de *threads* en cada nodo, algunos para ser usados por el usuario directamente y otros para ser imágenes que son necesarias para la migración. Los *threads* ejecutan una función que, en primera instancia, los bloquea en un semáforo local. Cuando el usuario crea un *thread* en un proceso particular, el sistema desbloquea uno de los *threads* en el *pool*, y reemplaza la función del *thread* con aquella que indicó el usuario. En los demás procesos, el sistema desbloquea al *thread* cuyo *stack* está en la misma dirección, pero en este caso se le suspende, convirtiéndolo en una imagen del original.

De esto se puede desprender que, respecto a la migración, un *thread* puede estar en dos estados: *activo* o *suspendido*. Cuando un *thread* está activo quiere decir que se encuentra en la cola de los *threads* `READY` o que está ejecutando. Por otro lado, si el *thread* está suspendido, está en la cola `SUSPENDED`. Esto es relevante ya que los *threads* en la cola `SUSPENDED` no pueden ser ejecutados. Cuando un *thread* se convierte en candidato a migración, el sistema lo coloca en la cola `SUSPENDED`, junto con los *threads* que están esperando recibir migraciones.

Para individualizar a los *threads* se les asigna un identificador. El *thread* en ejecución y sus imágenes tienen el mismo identificador; luego, es sencillo saber cuál será el *thread* que participa en la migración como destino. Cabe señalar que, en un momento determinado, habrá sólo un *thread* con el mismo identificador que esté en estado activo en todo el sistema; los demás estarán suspendidos.

Para migrar a un *thread* primero se obtiene la información del *stack*, utilizando las nuevas funciones de la biblioteca. En este paso, se requiere la dirección de inicio y el tamaño de la estructura que contiene el *stack*. Luego, se crea un mensaje en el que se copia el fragmento de memoria identificado anteriormente por su dirección de inicio y su tamaño. Este mensaje contiene también el identificador del *thread* a migrar y el del proceso origen. Una vez que se envía el mensaje al nodo destino, el *thread* es suspendido.

En el proceso destino, el sistema copia la estructura contenida en el mensaje, en la posición de memoria correspondiente a la estructura del *thread* imagen, el cual se encuentra suspendido. El *thread* imagen es fácilmente identificable, pues tiene el mismo identificador que el *thread* origen, señalado en el mensaje.

Finalmente, una vez que se ha hecho la copia de la estructura, el *thread* imagen es sacado de la cola `SUSPENDED` y reactivado, dejándolo en la cola de los `READY`. El proceso de migración se ilustra en el ejemplo de la Figura 2. El *thread* `t1` del proceso `P1` migra al proceso `P2`, sobre la imagen que había sido creada para él, durante la inicialización. El *thread* continúa su ejecución en el estado en que fue suspendido, y el *thread* original en `P1` se bloquea luego de la migración.

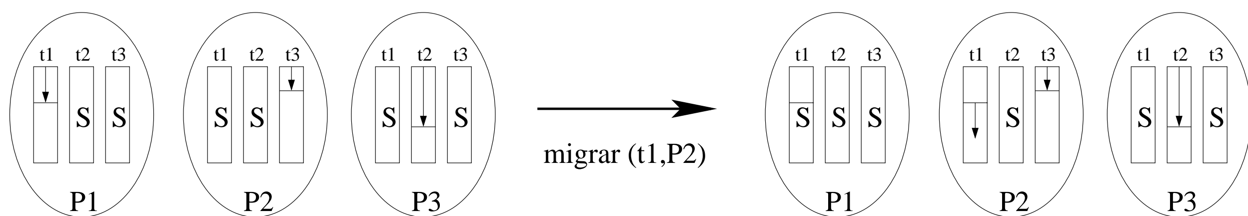

Figura 2: Migración de un *thread*

3.5. Costo de la migración

Puesto que el uso de migración de *threads* pretende aumentar la localidad de referencia, reduciendo los accesos remotos, es útil hacer una comparación del costo que tiene la migración de un *thread*, respecto a la migración de una página como resultado de un *page fault*.

Para los experimentos que llevamos a cabo, utilizamos computadores homogéneos con procesador Pentium III de 550 MHz y 128 MBytes de memoria RAM, ejecutando el sistema operativo RedHat Linux 9,0. Las comunicaciones fueron conmutadas a 100 Mbps.

En primer lugar, llevamos a cabo mediciones del tiempo que toma migrar un *thread* de un computador a otro. El *thread* retorna a su ubicación original inmediatamente, por lo que es posible estimar el tiempo de la migración en el punto de partida.

El Cuadro 1 muestra el tiempo que le toma al *thread* hacer la migración completa, volviendo a su origen. Asimismo, se muestra el tiempo que corresponde a cada trayecto, es decir, el tiempo total dividido por dos. En este caso, el *stack* de los threads se estableció en 4KBytes, lo que consideramos un tamaño razonable.

Cuadro 1: Tiempo de migración

Número de Ejecución	Ida y vuelta (μsegs)	Cada trayecto (μsegs)
1	4,672	2,336
2	4,716	2,358
3	3,971	1,986
4	4,039	2,020
5	3,087	1,544
Promedio	4,097	2,049

Como puede verse, en promedio, la migración de un *thread* demora aproximadamente 2 ms., desde el momento en que es suspendido en el origen, hasta que es reactivado en el destino.

Por otra parte, medimos el tiempo que demora la atención de un *page fault*, producido por la ausencia en forma local de una página de memoria. En este caso utilizamos un protocolo de consistencia secuencial, que se encarga de traer una copia de la página desde su ubicación actual, y activarla para su uso local. Las páginas son de 4KBytes.

El Cuadro 2 muestra el tiempo que toma capturar la interrupción producida por el *page fault*, el envío del requerimiento remoto por la página, el envío de la página, y su activación local. En esta configuración sólo se utilizaron 2 procesadores, para evitar el costo adicional de ubicar al dueño de la página, es decir, si la página no está en un nodo, con certeza estará en el otro.

Cuadro 2: Tiempo de atención de *page fault*

Número de Ejecución	Tiempo (μsegs)
1	2,267
2	2,364
3	2,360
4	2,327
5	2,301
Promedio	2,324

Puede verse que, en promedio, el tiempo de atención de un *page fault* es $2,3$ ms., desde el momento en que se produce la interrupción, hasta que el proceso del usuario puede continuar una vez que se ha obtenido una copia válida de la página.

Es importante resaltar lo similares que resultan los costos de migración para los *threads* y las páginas. Con una política de migración eficaz, la migración de *threads* puede minimizar la cantidad de transferencias de páginas entre los procesadores del sistema. Dado que el costo de la migración es bajo, resulta razonable pensar que el uso de este mecanismo puede resultar beneficioso.

4. Trabajos relacionados

En la literatura las distintas implementaciones para la migración de *threads* se distinguen en la manera que solucionan el problema del *stack*. De acuerdo a esto, se pueden identificar 3 alternativas. Primero están las implementaciones como Emerald [17], Arachne [18] y porch [19] que utilizan el apoyo del compilador y del lenguaje de programación para obtener más información de los punteros y así poder identificarlos y actualizarlos. Emerald [17] es un sistema y un lenguaje basado en objetos, diseñado para la construcción de programas distribuidos. Los objetos en Emerald, que contienen desde datos simples a procesos completos, pueden moverse libremente dentro del sistema, siendo ésta una de las principales características del lenguaje. Con la ayuda del compilador se describe el formato de las estructuras de datos y se apoya la traducción de punteros. Por su parte Arachne [18] provee migración de *threads* entre plataformas heterogéneas con manejo dinámico del tamaño del *stack*. En este sistema se agregan nuevas palabras claves al lenguaje C++ y utilizando un preprocesador se genera código escrito completamente en C++. Porch [19] es un prototipo de compilador que transforma programas escritos en C en otros, semánticamente equivalentes, pero que además son capaces de registrar y recuperar el estado de la ejecución utilizando técnicas de *checkpointing*. Está diseñado para funcionar en ambientes posiblemente heterogéneos. Este sistema está enfocado a migración de procesos y soporte a tolerancia a fallas. El problema de este tipo de soluciones es que hay que modificar el compilador o extender el lenguaje, lo que claramente puede traer problemas de portabilidad debido a la fuerte dependencia que se genera y la complejidad adicional que se incorpora.

Otras implementaciones, como por ejemplo, Ariadne [20] intentan identificar y actualizar los punteros en tiempo de ejecución. Este sistema es una biblioteca de *threads* a nivel de usuario que lleva a cabo la migración de threads utilizando las funciones de biblioteca del lenguaje C, `setjmp()` y `longjmp()`. En el nodo destino el *stack* migrado es inspeccionado para identificar y actualizar punteros. El problema de este enfoque es que es probable que no se detecten correctamente todos los punteros por lo que puede haber obvios errores de ejecución. Además, las direcciones de memoria que apuntan al *heap* no son actualizadas.

El tercer tipo de solución es el que se presenta en este artículo y en trabajos como Millipede [8], Nomad [21] o en Amber [22]. Millipede [8] es una implementación a nivel de usuario de un sistema de MCD con múltiples hebras y con migración transparente de procesamiento y páginas, diseñado para programación paralela sobre una red de computadores personales. Utiliza *threads* a nivel de sistema operativo (Windows-NT). Nomad [21] es un sistema de migración de *threads* diseñado para ser bastante liviano, ya que la migración en su inicio sólo implica mover el valor de los registros y sólo el comienzo de la página que contiene el *stack* del *thread*. El resto de la información es requerida y transportada después. Amber [22] es un sistema de MCD orientado a objetos que permite a una aplicación usar un red homogénea de computadores. Se soporta migración de datos y de *threads*. La ubicación de los objetos es manejada explícitamente por la aplicación. La principal desventaja es que necesitan eventualmente un espacio de direccionamiento muy grande y que, para el caso en que existen direcciones de memoria que no se encuentran dentro del *stack*, estos datos no son migrados junto con el *stack*.

Existen también soluciones que combinan ciertos aspectos de las anteriores. Es el caso de MigThread [16, 23, 24] que utiliza preprocesamiento para identificar zonas seguras para la migración o para apoyar el uso de la migración por parte del programador, y además maneja el *stack* y los registros en tiempo de ejecución. En esta implementación también se migra la información referenciada fuera del *stack*. Esta implementación está diseñada para funcionar en metacomputadores por lo que maneja *threads* a dos niveles: *threads* POSIX entre los nodos y *threads* de sistema operativo dentro de cada nodo.

5. Conclusiones y trabajo futuro

El uso de una biblioteca de *threads* a nivel de usuario facilitó en gran medida la implementación de nuestro mecanismo de migración. Con la incorporación de unas pocas funcionalidades adicionales para el manejo de los *stacks*, nuestra solución es simple pero robusta. Además, siguiendo el esquema modular de `DSM-PEPE`, el mecanismo se acopla de manera natural con el sistema MCD y en el futuro, con la política que conducirá el mecanismo de migración.

Los problemas más comunes que se presentan al implementar la migración de *threads* fueron abordados con soluciones simples. El esquema SPMD de `DSM-PEPE` ayuda a evitar conflictos con las direcciones del código y de las variables locales. El direccionamiento de las variables compartidas no presenta inconvenientes pues la consistencia de la memoria compartida distribuida en la que se almacenan, está garantizada por el sistema subyacente. La única restricción impuesta al programador se refiere a la prohibición de referenciar memoria fuera del espacio compartido y del *stack*.

Cuando se tenga el módulo encargado de la política de migración se deberá restringir el uso de la migración sólo a aquellos puntos dentro de la ejecución en que sea seguro migrar. Uno de los temas que quedan por desarrollar es el módulo que implementa la política de migración. Si bien este módulo no influye en la forma en que, en la práctica, se lleva a cabo la migración, sí es muy relevante para el completo aprovechamiento del mecanismo. Una vez que este módulo esté implementado, podremos efectuar estudios comparativos con otras implementaciones de mecanismos de migración de *threads* y, en particular, estudiar el verdadero impacto de la migración en el sistema de memoria compartida distribuida.

Dentro de los análisis que nos interesa completar se encuentra un estudio sobre el efecto que tiene la migración de *threads* en diversos tipos de aplicaciones bajo diversos protocolos de consistencia de memoria relajados, como es el caso de *entry consistency* [25] y *lazy release consistency* [26].

Referencias

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

[2] 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, August 1997.

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

[4] David Mosberger. Memory Consistency Models. Technical Report TR 93/11, Department of Computer Science, University of Arizona, 1993.

[5] S. Adve and K Gharachorloo. Shared memory consistency models: A tutorial. Technical Report ECE-9512, Rice University, Houston, TX (USA), 1995.

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

[7] Kritchalach Thitikamol and Peter J. Keleher. Thread Migration and Communication Minimization in DSM Systems (Invited Paper). *Proceedings of the IEEE*, 87(3):487–497, March 1999.

[8] Ayal Itzkovitz, Assaf Schuster, and Lea Wolfovich. Thread Migration and its Applications in Distributed Shared Memory Systems. Technical Report LPCR9603, Technion, Israel, July 1996.

[9] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

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

[11] Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software – Practice and Experience*, 21(8):757–785, August 1991.

[12] E. R. Zayas. Attacking the Process Migration Bottleneck. In *11th Symposyum on Operating Systems Principles (SOSP'87)*, pages 13–24, 1987.

[13] Federico Meza, Alvaro E. Campos, and Cristian Ruz. On the Design and Implementation of a Portable DSM System for Low-Cost Multicomputers. In *International Conference on Computational Science and Its Applications, ICCSA 2003*, number 2667 in Lecture Notes in Computer Science, pages 967–976, Montreal, Canada, May 2003. Springer-Verlag.

[14] Alvaro E. Campos and Federico Meza. DSM-PEPE: Un Sistema de Memoria Compartida Distribuida para Multicomputadores de Bajo Costo. In *Anales del VIII Congreso Argentino de Ciencias de la Computación, CACIC 2002*, pages 205–216, Buenos Aires, Argentina, October 2002. Article-C163.

[15] Gordon V. Cormack. A micro-kernel for concurrency in C. *Software—Practice & Experience*, 18(5):485–491, May 1988.

[16] Hai Jiang and Vipin Chaudhary. MigThread: Thread Migration in DSM Systems. In *Proceedings of the ICPP Workshop on Compile/Runtime Techniques for Parallel Computing*, August 2002.

[17] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[18] Bozhidar Dimitrov and Vernon Rego. Arachne: A Portable Threads Library Supporting Migrant Threads on Heterogeneous Network Farms. In *Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 102–114, 1997.

[19] Volker Strumpen. Compiler Technology for Portable Checkpoints. Submitted for publication, 1998.

[20] Edward Mascarenhas and Vernon Rego. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. *Software – Practice and Experience*, 26(3):327–356, March 1996.

[21] Scott Milton. Thread Migration in Distributed Memory Multicomputers. Technical Report TR-CS-98-01, Dept of Comp Sci & Comp Sciences Lab, Australia National University, Canberra 0200 ACT, Australia, February 1998.

[22] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park AZ USA, 1989.

[23] Hai Jiang and Vipin Chaudhary. Compile/Run-time Support for Thread Migration. In *16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida, April 2002.

[24] Hai Jiang and Vipin Chaudhary. On Improving Thread Migration: Safety and Performance. In Sartaj Sahni, Viktor K. Prasanna, and Uday Shukla, editors, *Proceedings, 9th International Conference on High Performance Computing — HiPC2002*, volume 2552 of *Lecture Notes in Computer Science*, pages 474–484, Berlin, Germany, December 2002. Springer-Verlag.

[25] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, Pittsburgh, PA (USA), 1991.

[26] Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *19th Annual International Simposium on Computer Architecture, ACM*, pages 13–21, May 1992.