Reinventing Scheduling for Multicore Systems

CARLOS MARTÍN FLORES GONZÁLEZ, Carné: 2015183528

Instituto Tecnológico de Costa Rica
Maestría en Computación
Sistemas Operativos Avanzados
Profesor: Francisco Torres Rojas, Ph.D

Los *Schedulers* en los sistemas operativos (SO) de hoy tiene el objetivo principal de mantener todos los núcleos ocupados ejecutando algún hilo(*thread*). El uso de memoria *on-chip* no es planificado(*scheduled*) explícitamente: el uso de un thread de algún dato mueve implícitamente el dato del núcleo de caché local. Esta *scheduling* implícito de memoria *on-chip* a menudo trabaja bien, pero puede ser ineficiente para datos de lectura/escritura compartidos entre múltiples threads o para datos que son muy grandes para caber en el caché de un núcleo. Para datos de lectura/escritura compartidos, los mensajes de coherencia de caché pueden saturar las interconexiones del sistema para algunas cargas de trabajo. Para conjuntos de datos grandes, el riesgo es que cada dato pueda ser replicado en muchos cachés. Se propone el uso de un *scheduler* que asigne objetos de datos a cachés *on-chip* y migre threads entre todos los núcleos conforme van accesando los objetos, en una manera similar a los sistema NUMA que migran threads entre nodos. Esta migración puede disminuir el acceso a la memoria, debido a que lleva a los threads cerca de los datos que usan, asi como disminuir la duplicación de datos entre múltiples núcleos de caché, permitiendo que datos aún más distintos se pongan en caché. Un *scheduler* que mueve operaciones a objectos es un O^2 *scheduler*.

O^2 ***Scheduling***. Escenario: una carga de trabajo en donde cada thread busca repetidamente un archivo en un directorio seleccionado aleatoriamente. Cada directorio es un objeto y cada búsqueda es una operación.

En un sistema con cuatro núcleos, un *scheduler* basado en threads va a programar cada thread a un núcleo. Cada núcleo va a poner en caché directorios usados recientemente. Si el conjunto de trabajo es menor que el tamaño de un núcleo de caché, el rendimiento estará bien. Si el conjunto de trabajo es más grande que un núcleo de caché, entonces todos los threads probablemente van a pasar mucho tiempo esperando por *on-chip* DRAM. Por otro lado, un O^2 *scheduler* particionará los directorios a través de todos los cachés para tomar ventaja de la memoria total *on-chip*. El O^2 *scheduler* migrará cada búsqueda hacia el núcleo que mantiene en caché el directorio relevante. Si el conjunto de trabajo de directorios es más grande que un núcleo de caché y el costo de migración es relativamente bajo, el O^2 *schedule* ve a proveer mejor rendimiento que el *scheduler* basado en threads.

Challenges. (1) Un O^2 *scheduler* debe balancear tanto objetos como operaciones a través de cachés y núcleos. No debería de asignar más objetos que los que quepan en el núcleo de caché o dejar algunos núcleos inactivos mientras otros están saturados. (2) El *scheduler* debe entender lo suficiente acerca de la carga de trabajo para planificarla bien, tien que estar en la capacidad de identificar objetos y operaciones, encontar tamaños de objetos y estimar tiempos de ejecución de las operaciones. (3) El *scheduler* debe de controlar cómo los objetos son almacenados en cachés. (4) El *scheduler* necesita una forma eficiente de migrar threads.

CoreTime. CoreTime es un diseño de O^2 *scheduler* que opera como una librería de tiempo de ejecución para programas en C. *Interface:* CoreTime depende de los programadores de aplicación para especificar lo que tiene que ser planificado. CoreTime provee dos anotaciones de código con la que los programadores marcan el inicio y el final de una operación. Toman un argumento que especifica la dirección que identifica el objeto. `cb_start`(*o*) realiza una búsqueda en una tabla para determinar si el objecto *o* está planificado por un núcleo en específico. Si la tabla no contiene un núcleo para la operación *o* se ejecuta localmente, de otra manera el thread se migra al núcleo retornado en la búsqueda. `cb_start` añade automáticamente un objeto a la tabla si el objecto es muy costoso de recuperar(*fetch*). Las anotaciones de CoreTime provistas por los desarrolladores ayudan a reducir el número de objetos que CoreTime considera para planificación. *Algorithm:* CoreTime usa un algoritmo voraz *first fit "cache packing"* para decidir qué núcleo asignar a un objeto. Está motivado en la observación que migrar una operación para manipular algún objeto *o* es únicamente beneficioso cuando el costo de la migración es menor el costo de recuperar(*fetching*) el objeto de la DRAM o algún caché remoto. El algoritmo *cache packing* trabaja asignando cada objeto que es costoso de recuperar a un caché con espacio libre. El algoritmo se ejecuta en un tiempo $\Theta(n \log n)$, donde n es el número de objetos. *Cache packing* podría asignar varios objetos populares a un solo núcleo y los threads van a esperar para operar en los objetos. *Runtime monitoring:* CoreTime usa contadores de eventos de AMD para detectar objetos que son costosos de recuperar y que deberían ser asignados a un núcleo. Para cada objeto, CoreTime cuenta el número de intentos fallidos de caché que ocurren entre un para de anotaciones CoreTime y asumo que los fallidos son causados por la recuperación del objeto. CoreTime también usa contadores de eventos de hardware para detectar cuando demasiadas operaciones son asignadas a un núcleo o si demasiados objetos son asignados al caché. CoreTime lleva registro del número de ciclos inactivos(*idle*), cargas desde la DRAM y cargas desde el caché L2 para cada núcleo. Si un núcleo está raramente inactivo o si a menudo realiza cargas de la DRAM, CoreTime va a mover periódicamente una porción de los objetos desde ese núcleo de caché hacia el núcleo de caché que tiene mas ciclos inactivos y que raramente realiza cargas desde el caché L2. *Migration:* Cuando un thread llama a `ct_start`(*o*) y *o* se asigna a otro núcleo entonces CoreTime va a migar el thread. El núcleo que el thread se está ejecutando guarda el contexto del thread en un *buffer* compartido, continua la ejecución de otros threads haya en su cola y establece una bandera que el núcleo destino consulta periódicamente. Cuando el núcleo destino nota una migración pendiente carga el contexto del thread y continua ejecutando. Eventaualmente el thread va a llamar a `cb_end`, que salga el contexto del thread y establece una bandera que indica al núcleo original que la operación está completa y el thread está listo para correr en otro núcleo. *Implementation:* CoreTime corre en Linux pero puede ser portado a otros SO similares. CoreTime crea un `pthread` por núcleo, atado al núcleo con `sched_setaffinity()`, y establece la prioridad de *process scheduling* lo más alto posible usando `setpriority()` para evitar ser des-planificado por el kernel.

Preliminary Evidence. Hardware: Sistema AMD, 4 chips Opteron quad-core de 2GHz. Cada núcleo con su propio caché L1 y L2. Los 4 chips comparten un caché L3. Latencias: L1 3 ciclos, L2 14 ciclos y L3 75 ciclos. Latencias de recuperación remotas varían de 127 a 336 ciclos para recuperar el banco DRAM más distante. El costo de migración en CoreTime es 2000 ciclos. *Setup:* Se midió el rendimiento de CoreTime cuando se aplicó al sistema de archivos usando dos directores de búsqueda de referencia. El sistema de archivos de derivaba de una implentación EFSL FAT y se modificó para usar operaciones de memoria en lugar de disco, esto para no usar caché de *buffer* y para tener un mayor rendimiento en las búsquedas anidadas de nombres de archivos. Se concentró en búsqueda de directorios, añadiendo *spin-locks* por directorio y anotaciones CoreTime. Cada directorio es un objeto CoreTime y cada búsqueda de nombre de archivo una operación. Las cargas de trabajo involucraban un thread en cada núcleo búscando repetidamente un archivo seleccionado aleatoriamente de un directorio seleccionado aleatoriamente. Cada directorio contenía 1000 entradas y cada entrada usaba 32 bytes de memoria. Podría esperarse que CoreTime mejore el rendimiento de estas cargas de trabajo cuando el conjunto total de entradas de directorios es lo suficientemente grande para no caber en un solo núcleo de caché. Resultados:

Se mide el rendimiento de la comparación del sistema de archivos cuando se selecciona aleatoriamente un nombre de archivo usando una distribución uniforme. Para tamaños de datos totales entre aproximadamente 512Kb y 2Mb, una copia completa del conjunto total de directorios puede caber en cada uno de los los cachés L3 de los 4 chips AMD. Con y sin CoreTime hay buen rendimiento debido a que todos las búsquedas operan en dados de caché local. Cuando la cantidad total de datos es notablemente más grande que 2Mb, una copia completa no puede caber en cada uno de los caché L3 de los chips AMD y el rendimiento con CoreTime es dos o tres veces más rápido que sin CoreTime. CoreTime automáticamente asigna directorios a los cachés cuando se detectan fallos de caché durante búsquedas. Sin CoreTime, los núcleos tiene que leer los contenidos del directorio desde la DRAM o cachés remotos. Con CoreTime, no hay duplicación de datos en el cache y cada búsqueda es ejecutada en el núcleo que tiene el directorio en su caché. La cantidad toal de espacio de caché es 16Mb, entonces CoreTime puede evitar usar la DRAM hasta que haya más de 16Mb de datos. El rendimiento se viene abajo andate de ese punto y más data es almacendada en la L3 en lugar de la L2. CoreTime está en la capacidad de rebalancear directorios a través de cachés y rinde más de el doble de rápido para la mayoría de tamaños de datos que sin CoreTime.

Discussion: *Future Multicores:* se espera que *multicores* futuros puedan ajustarse a problemas como: anchos de banda de memoria *off-chip* elevados, alto costo para migrar un *thread*, el poco tamaño agregado de memoria *on-chip* y la limitada habilidad del software de controlar cachés de hardware. Esto para favorecer O^2 *scheduling*. Además, el aumento en el número de instrucciones de CPU que le permiten al software controlar el caché es una evidencia que los fabricantes de chips reconocen la importancia de permitir al software controlar el comportamiento del caché. Estas tendencias van a dar como resultado procesadores en donde el O^2 *scheduling* podría ser atractivo para un gran número de cargas de trabajo. Los *multicores* podrían tener núcleos heterogeneos lo que puede complicar el diseño de un O^2 *scheduler*. Los procesadores podrían no tener coherencia de caché y podrían depender del software para gestionar la colocación de los objetos. Si este fuera el caso entonces el O^2 *scheduler* tiene que estar involucrado en esta colocación. También si mensajes activos son soportados por hardware esto podría reducir el *overhead* de la migración. *O^2 Improvements:* El algoritimo de O^2 *scheduling* está en fase preliminar y podría beneficiarse de *object clustering:* si un thread u operación usa dos objetos simultáneamente entonces podria ser mejor poner ambos objetos en el mismo caché si es que caben. También hay algunas áreas no exploradas en el algoritmo de O^2 *scheduler*. Por ejemplo, algunas veces es mejor replicar objetos de solo lectura y otras podría ser mejor planificar objetos distintos. Conjuntos de trabajo más grandes que el total de memoria *on-chip* presentan algo interesante. En estas situaciones O^2 *schedulers* podrían usar una política de reemplazo de caché que, por ejemplo, guarde los objetos accesados más frecuentemente *on-chip* y guarde los menos accesados menos frecuentemente *off-chip*. Para usar O^2 *scheduler* como el *scheduler* del sistema de por defecto, el O^2 *scheduler* tiene que llevar registro qué procesos posee un objeto y sus operaciones. Con esta información el O^2 *scheduler* podría implementar prioridades y justicia(*fairness*). El soporte del compilador podría reducir el trabajo de los programadores y proveer al O^2 *scheduler* con más información.

1. ¿CUÁL ES EL PROBLEMA QUE PLANTEA EL *PAPER*?

Los *schedulers* tradicionales se concentran en mantener unidades de ejecución ocupadas al asignar a cada núcleo un thread para correr. Sin embargo, los *schedulers* deberían de enfocarse en la alta utilización de memoria *on-chip* en lugar de núcleos de ejecución para reducir el impacto de los accesos costosos a DRAM y los accesos a cachés remotos.

2. ¿POR QUÉ EL PROBLEMA ES INTERESANTE O IMPORTANTE?

Conforme el número de núcleos por chip crece, los ciclos de computación continuaran creciendo relativamente más abundante que el acceso a memoria *off-chip*. Para lograr buen rendimiento, las aplicaciones necesitarán hacer

un uso más eficiente de memoria *on-chip*. Memoria *on-chip* probablemente continuará llegando en la forma de muchos cachés pequeños asociados con núcleos individuales. El reto centro será administrar estos cachés para evitar accesos a memoria *off-chip*

3. ¿QUÉ OTRAS SOLUCIONES SE HAN INTENTADO PARA RESOLVER ESTE PROBLEMA?

4. ¿CUÁL ES LA SOLUCIÓN PROPUESTA POR LOS AUTORES?

Se argumenta que la solución requiere de un nuevo enfoque de planificación(*scheduling*), uno que se concentre en assignación de objetos de datos a los núcleos de los cachés en lugar de asignar threads a los núcleos. Se propone el uso de un *scheduler* que asigne objetos de datos a cachés *on-chip* y migre threads entre todos los núcleos conforme van accesando los objetos, en una manera similar a los sistema NUMA que migran threads entre nodos. Esta migración puede disminuir el acceso a la memoria, debido a que lleva a los threads cerca de los datos que usan, asi como disminuir la duplicación de datos entre múltiples núcleos de caché, permitiendo que datos aún más distintos se pongan en caché. CoreTime *scheduler* que se propone se basa en O^2 *Scheduling*.

5. ¿QUÉ TAN EXITOSA ES ESTA SOLUCIÓN?