**OS Scheduling with NEST: Keeping Tasks Close Together on Warm Cores**

**Abstract**

Para dar soporte a aplicaciones paralelas, el planificador CFS distribuye las tareas a los cores, pero esto se ha observado que, en entornos de servidores no resulta eficiente (porcentaje tiempo invertido en la planificación puede llegar a ser elevado respecto al tiempo de ejecución) (tareas asignadas a cores de bajas frecuencias). Se propone la reutilización de cores, construyendo un “nido” de cores para utilizar con prioridad en tareas de planificación.

Rendimiento y consumo de energía similar a CFS en tareas altamente paralelizables, pero mejor si hay aplicaciones con menos tareas que cores.

**Introduction**

Principal tarea de un SO 🡪 Alojar tareas en cores de forma que se maximiza el rendimiento de la aplicación.

Propiedad work conservation 🡪 Si una tarea es alojada en un core que no está idle, entonces no puede haber ningún core idle

A la hora de alojar las tareas en los cores, se debe tener en cuenta su frecuencia, para que estas puedan acceder a los recursos de forma eficiente. CFS no tiene en cuenta la frecuencia a la hora de repartir las tareas, pero las CPUs modernas tienen cores de distintos rendimientos, por lo que hay que ajustar la distribución para alcanzar el máximo rendimiento.

Los cores pueden ajustar su frecuencia de forma independiente 🡪 Colocar las tareas en los cores de altas frecuencias.

Servidores más accesibles que pueden ser usados tanto para computación de alto rendimiento tradicional (las aplicaciones se dividen en tareas pequeñas), como recursos computacionales para aplicaciones demandantes de CPU y memoria o disco. Los cores requeridos pueden variar, por lo que el planificador debe realizar una asignación lo más óptima posible.

La cores ofrecen frecuencias turbo, que sobrepasan los hercios base. Para lograr esto se deben tener en cuenta varios factores. Compromiso Hardware – Software. El software oferta unos rangos de frecuencias y es el hardware el que selecciona una de ellas para el core. Depende del número de cores utilizados en el socket. Se maximiza la frecuencia minimizando el número de cores del socket.

Propuesta de planificador: NEST. Asignar las tareas a un conjunto de cores utilizados recientemente. Se introduce un proceso idle en los cores idle recientes para conservar la alta frecuencia. Cuando una tarea es expulsada, se recuerda en que core ejecutaba para tratar de volver a asignarla al mismo cuando necesite ejecutar de nuevo.

Aplicaciones secuenciales o aplicaciones con tantas tareas como cores 🡪 Rendimiento similar a CFS

Aplicaciones con muchas tareas concurrentes y mucha planificación de tareas (fork, block, etc) 🡪 Mejor rendimiento que CFS

Este artículo se basa en 2 principios:

* Keep tasks close together: Minimizar el número de cores empleado y tratar siempre de reutilizarlos
* Keep cores warm: Mantener los cores ocupados para conservar la frecuencia

Con NEST se consigue tanto mejorar el rendimiento como reducir el consumo de energía de la CPU.

**Background: Linux CFS and Power Governors**

Planificador NEST extiende del planificador CFS de Linux. CFS se basa en la heurística a la hora de realizar la planificación (Distintas fases: task fork, task exec, task wakeup, load balancing).

NEST tiene un mayor impacto en fork y wakeup (1300 líneas de las 11000 de CFS).

Organización de los cores en niveles: NUMA (cores en la misma máquina) 🡪 SMP (Cores en el mismo die) 🡪 SMT (hyperthreads compartiendo el mismo core físico). Cada core se identifica con una tupla de estos grupos.

Fork: Se analiza la carga de forma descendente (Primero el grupo, luego los cores dentro del grupo, etc). La carga viene determinada por diferentes factores (número de cores idle, carga reciente de los cores, tiempos previstos de wakeup y las preferencias de NUMA). Los cores están numerados y la búsqueda se realiza en orden, módulo el número de cores. Esto se repite en todos los niveles. El hecho de buscar linealmente por los cores, impide que se favorezca la reutilización, ya que si otro core idle va antes, será seleccionado.

Wakeup: CFS selecciona un target core, que puede ser o bien el core que había utilizado la tarea, o bien el core que ha hecho el wakeup. La decisión entre estos 2 se realiza con heurísticas. Después CFS busca un core idle en el die objetivo. Se busca un core que tenga tanto el core como el hyperthread idle. Si no lo encuentra, busca un core idle entre un grupo reducido. Si falla, se comprueba si el hyperthread del target core está idle. Si también falla, CFS selecciona el target.

CFS no favorece la conservación del trabajo ya que solo busca en un die y con pocos esfuerzos para encontrar un core en ese die. Nuevamente, para la elección se recorren los cores linealmente por lo que se desfavorece la reutilización de cores.

SMove Scheduler (Problema de inversión de frecuencia): Ocurre cuando un proceso padre hace un fork o un wakeup sobre un proceso hijo, y queda a la espera de los resultados de la tarea. Normalmente ocurre que el padre está ejecutando en un core de mayor frecuencia, y el hijo comienza a ejecutar en un core idle. El hijo ejecuta en un core de menor frecuencia y el padre queda a la espera de los resultados dejando disponible un core de mayor frecuencia. Se propone el planificador SMove que coloca el proceso hijo en el core del proceso padre para beneficiarse de la alta frecuencia. Esta asignación solo se realiza si, el proceso hijo está corriendo en un core de baja frecuencia, y se inicia un timer para asignar la tarea hijo en el core seleccionado por CFS si esta no se ha podido planificar en el core de la tarea padre. De esta forma evitamos posibles delays debidos a que, por ejemplo, la tarea padre no se bloquee inmediatamente.

Linux Power Governors: El planificador desconoce acerca de frecuencias. La frecuencia de los cores viene dada por una mezcla de los Linux Power Governors y el hardware. Los Powers governors proponen un rango de frecuencias a emplear, y es el hardware el que determina cual en base a diferentes observaciones. Cuando se hace referencia a planificadores, también deben tenerse en cuenta los power governors.

En Linux son *Performance* y *Schedutil*.

*Performance* propone usar la frecuencia nominal de la máquina y es el propio hardware el que elige si usar esta o la turbo. Proporciona alto rendimiento, pero descuida el ahorro de energía que podría resultar de poner tareas ligeras a baja frecuencia.

*Schedutil* tiene en cuenta el planificador con las tareas recientes para buscar un equilibrio entre rendimiento y ahorro energético. Permite usar todo el rango de frecuencias. Si ve que una tarea ha utilizado mucha CPU recientemente, propone utilizar una frecuencia superior.

**The NEST approach**

La idea principal reside en definir un conjunto de cores que han sido utilizados recientemente y considerarlos alta prioridad a la hora de asignar tareas. Limitar el número de cores a utilizar, hace que se seleccionen frecuencias altas. Diseñar un planificador que tenga en cuenta esto a la hora de planificar. Si el set de cores es demasiado grande, las tareas se dispersarán al igual que en CFS; y si es muy pequeño, todas las tareas querrán ir a los mismos cores por lo que habrá una sobrecarga.

Se definen dos sets (primary y reserve) para la asignación de tareas. En el primero de ellos se colocan cores que están en uso o que han sido usados recientemente y se espera que sean reutilizados pronto. En el segundo set se colocan los cores que estaban en el primer set y no han sido utilizados recientemente, o han sido utilizados por CFS.

A la hora de hacer la asignación, en primer lugar, se busca en el primary nest un core libre. Si no existe, se busca en el reserve nest. Y si en el segundo set tampoco existe un core libre, entonces se realiza la asignación normal con CFS. La búsqueda en el primary nest comienza en el core que había utilizado la tarea anteriormente (o en el core del padre si se trata de un fork). La búsqueda en el reserve nest comienza en un core fijo (normalmente el core que ejecutó la llamada al sistema para iniciar NEST). La búsqueda considera primera los cores del mismo die, creando la posibilidad de dejar dies sin usar y por lo tanto ahorrando energía. A diferencia de CFS, NEST puede seleccionar cualquier core idle sin tener en cuenta la carga reciente de trabajo, ni la actividad de los hyperthreads.

¿Cómo se mueven los cores entre nests? Si se selecciona un core de reserve nest, este pasa a primary nest. Si se elige un core de CFS, normalmente, se sitúa en reserve nest. El número de cores en los nest está limitado por Rmax. Si se supera ese número, entonces el core seleccionado por CFS no se añade a reserve nest. Del mismo modo, si un core del primary nest lleva mucho tiempo sin usarse (supera el umbral Premove), este pasa a reserve nest, y si está lleno es directamente descartado. Cuando una tarea finaliza su actividad en un core, este ya no es considerado útil y se degrada del primary nest a reserve nest.

Puede ocurrir que muchas tareas estén intentando utilizar el primary nest, pero no encontrar alojamiento ya que el core que habían utilizado anteriormente se encuentra ocupado por otra tarea. Se establece un número máximo de intentos (RImpatient), para conseguir ese core antes de marcar esa tarea como impaciente y posibilitar que el core seleccionado por CFS pase directamente al primary nest.

El objetivo es mantener los cores a alta frecuencia, y es sabido que las tareas paran brevemente para sincronización o I/O. Estas pausas pueden reducir la frecuencia y es por ello que se modifica el proceso idle para que ejecute durante un breve periodo, comprobando si alguna tarea ejecuta en el hyperthread, y si esto ocurre se detiene dejando el espacio para la tarea.

Lo ideal es que una tarea vuelva a ejecutar en el core en el que lo había hecho previamente, pero si esto no es posible, se debe buscar otro core, lo que puede afectar a otra tarea, provocando así una cascada de migraciones. NEST, además de recordar el core usado previamente, también recuerda el anterior (es decir 2 hacia atras). Si ambos coinciden, entonces la tarea está ligada a ese core, y siempre se tratará de planificar en el si está en el primary nest y libre.

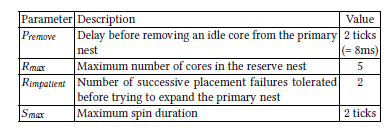
El primary nest es más efectivo cuando su número de cores se corresponde con el número de tareas ejecutando concurrentemente. CFS únicamente considera, para la búsqueda de cores libres, el die anterior en el que ejecutaba o el die que ha reanudado la tarea, mientras que NEST contempla todos los dies.

En Linux la asignación de tareas se realiza en 2 pasos: primero se selecciona el core y a continuación se encola la tarea en su running queue. El hecho de no existir un lock global hace que 2 tareas puedan seleccionar el mismo core a la vez, lo que impacta negativamente en el rendimiento. Esto ocurre en CFS, y en NEST es todavía peor ya que el número de cores es más reducido. Por ello, se asocia una flag a cada running queue, indicando si una tarea ya ha sido asignada al core correspondiente. Se utiliza una instrucción compare-and-swap (atómica). Esto también se puede aplicar a CFS, pero el beneficio no sería tanto como en NEST.

**Implementation**

Para la implementación de este planificador se han añadido unas 500 líneas de código, mayoritariamente en ficheros de CFS (kernel/sched/fair.c) y el planificador core (kernel/sched/core.c). Se utiliza una llamada al sistema existente para indicar que una tarea y todos sus hijos deben ser planificados con NEST.

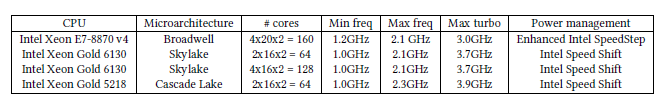
Para los diferentes parámetros mencionados anteriormente, se han utilizado los siguientes valores:

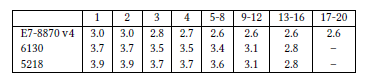


**Evaluation**

Para evaluar el rendimiento del planificador NEST, se debe probar con una gran variedad de programas y benchmarks, que empleen diferentes números de cores y tareas. Se comprueba tanto su rendimiento como su consumo energético.

Las pruebas se basarán en el planificador CFS v5.9 de Linux usando el schedutil power governor.

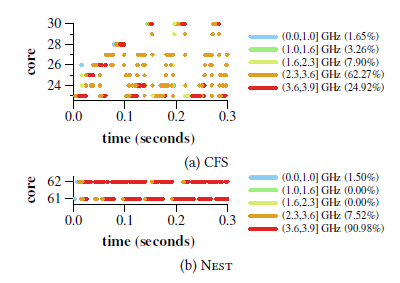
Hardware: Se emplean 3 generaciones de Intel Xeon (de 2016 a 2019). Un die coincide con un socket.



Measurements: Para los test de rendimiento, 2 warm-up y 10 ejecuciones. El speedup se calcula dividiendo el tiempo con NEST entre el tiempo con CFS-schedutil. Las comparaciones están normalizadas de forma que un 0 significa igual rendimiento. Se muestra un umbral del +-5% para mostrar las grandes mejoras o degradaciones. El consumo de energía se mide con turbostat, realizando mediciones cada 0,5 segundos.

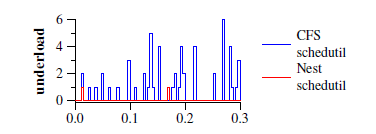
La configuración del software está basada en shell scripts. Se ejecutarán cientos o incluso miles de tareas. La situación ideal para NEST es ejecutar muchas tareas cortas y que ejecutan solas.

Caso de estudio: LLVM usando cmake para compilación utilizando Ninja



CFS dispersa las tareas en 8 cores, provocando que las frecuencias se vean reducidas, mientras que NEST únicamente emplea 2 cores, manteniendo la frecuencia en un valor alto.

Se define el concepto de underload, que es la diferencia entre el número de cores usado y el número de tareas que están disponibles para ejecutar simultáneamente.



Un valor positivo en esta métrica, indica baja reutilización de cores, ya que se ha utilizado otro que llevaba más tiempo sin usarse, lo que provoca una bajada de rendimiento.

Performance análisis: Se utilizan los scripts disponibles en Phoronix Timed Code Compilation Test (conjunto de tests que miden la cantidad de tiempo necesaria para construir diferentes paquetes software open source). Esta suite utiliza diferentes compiladores como pueden ser gcc, llvm\_ninja, o php. Solo se tiene en cuenta el software cuyos scripts de configuración son generados con autotools o cmake, ya que separan muy bien la fase de configuración, del resto de la construcción.

Figura 4 muestra el underload por segundo.

Figura 5 muestra los speedups conseguidos con NEST.

Figura 6 muestra las frecuencias de los cores al ejecutar los scripts.

Con NEST se reduce el número de cores utilizados y, por lo tanto, los cores que corren tareas están altamente empleados, usando las frecuencias más altas prácticamente siempre.

Conclusión de las gráficas: En la mayoría de escenarios NEST consigue un mayor rendimiento porque CFS usa demasiados cores, haciendo que estos no funcionen a las frecuencias más altas.

Lo primero que nos podría pasar por la cabeza al pensar en la utilización de altas frecuencias, es que el consumo energético aumentará. Esto es cierto, pero se debe tener en cuenta que, al tener una mayor frecuencia, las tareas tardan menos en ejecutar, por lo que el tiempo de uso disminuye y el consumo asociado también, contrarrestándolo.

Figura 7 muestra los consumos de energía comparando CFS y NEST.

Impact of NEST features: Se repiten determinados tests pero modificando algún parámetro. La modificación principal es eliminar el reserve nest, dejar únicamente el primary nest, y ver como impacta esto en el rendimiento. Se observan degradaciones del rendimiento de en torno a un 5% en las máquinas 6130 y 5218, y de un 16% en la E7-8870. El beneficio que aporta la reserve nest es mantener la primary nest con un tamaño reducido, pero permitiendo algunos cores extra permanece idle durante un tiempo muy reducido.

Comparison with Smove: Este planificador consigue mejores resultados que CFS normal, pero aun así queda bastante por detrás de NEST.

DaCapo benchmark suite: Esta suite de tests se realiza sobre una serie de aplicaciones Java. Para conseguir resultados más realistas se emplean técnicas para evitar los efectos del JIT compiler (compilación en tiempo real) y el recolector de basura. DaCapo solo considera la última de una serie de N ejecuciones, pero para estas pruebas se coge todo el rango para tener más opciones y la desviación de los resultados es apenas del 2%.

Las pruebas emplean desde aplicaciones con un número pequeño de tareas, hasta aplicaciones multicore potentes. Los resultados con NEST varían, teniendo desde un 6% de degradación con aplicaciones de una tarea, hasta una mejora del 40%.

Un ejemplo de ejecución es h2 (una de las apps de la suite). Con el schedutil power governor, al utilizar CFS, las tareas se dispersan sobre los cores de un socket, haciendo que estos no consiguen las máximas frecuencias posibles (el 32% en el rango 3.1 – 3.4 GHz y el 3% en el rango 3.4-3.7 GHz), mientras que al realizar la prueba con NEST, las tareas emplean en total unos 10 cores, posibilitando el aumento de frecuencias (el 57% en el rango 3.1 – 3.4 GHz y el 13% en el rango 3.4-3.7 GHz). Esto nos proporciona un speedup de en torno al 20%. Sin embargo, si esta prueba la repetimos con el performance power governor, los resultados son más parejos, puesto que siempre se tienden a utilizar las turbo-frecuencias. Aun así, NEST consigue utilizar mayores frecuencias puesto que tiende a minimizar el número de cores empleado.

Impact of NEST features in DaCapo: Se consideran modificaciones de NEST sobre spinning, nest compaction y reserve net. El mayor impacto supone la eliminación de spinning (degradación de en torno a un 20%). Eliminar la nest compaction, hace que las aplicaciones se dispersen más entre los cores, reduciendo su rendimiento. A diferencia de la anterior suite de tests, aquí la reserve nest no tiene tanto impacto ya que lo primary nest se construye adecuándose bien a las tareas, sin tener la necesidad de acudir a la reserve nest.

NAS Parallel Benchmarks (Colección de HPC kernels): Estos tests, asignan una tarea por core. Se asigna en tiempo de fork y permanece en el core hasta el final de su ejecución. Sobrecargar cores, dejando otros sin utilizar puede provocar retardos, por lo que NEST debe conseguir una asignación optima.

En este caso, no hay tanta diferencia entre CFS y NEST para las máquinas con 2 sockets, ya que, prácticamente no hay movimiento entre ellos y por ende no se pueden conseguir altas frecuencias.

Sin embargo, para la máquina con 4 sockets, los speedups conseguidos son altos con ambos power governors, aunque los resultados no son demasiado precisos ya que CFS obtiene una desviación de en torno al 54%.

La característica de NEST que más impacto tiene en este caso es el favorecimiento de los cores recientemente utilizados, mientras que las demás (spinning, reserve nest o compaction), prácticamente no tienen efecto.

Phoronix multicore suite: 90 benchmarks con un total de 222 tests. La mayoría de los tests no se ven afectados fuertemente en cuanto a rendimiento por la elección del planificador. En más del 90% de los casos, la ganancia o pérdida del rendimiento se inferior al 5,2%, a excepción de la máquina E7-8870 que consigue mejoras superiores a ese valor en el 21% de los tests.

Zstd compression 7 and 10: Con CFS las tareas se dispersan más entre los cores, haciendo que las frecuencias sean bajas, pero al tratarse de tareas que ejecutan durante poco tiempo, el impacto no es muy grande. Con NEST el número de cores se reduce a unos pocos del mismo socket, pero de igual forma, consiguiendo buenos speedups pero no tan impactantes como en otros benchmarks, por el hecho de ser tareas de poca duración.

Rodinia 5: En este caso casos los resultados son contradictorios en función de la máquina empleada. Por ejemplo, con el hardware 6130 y el 5218, se consiguen speedups positivos usando NEST frente a CFS-performance, pero en el hardware E7-8870, lo que ocurre con NEST es una degradación del 8%. Esto se debe a que Rodinia 5 emplea principalmente 36 cores; Con CFS estos son todos del mismo socket, mientras que con NEST, para favorecer la conservación del trabajo, estos están desperdigados por toda la máquina. Con CFS-schedutil, las tareas comparten cores físicos mientras que, con NEST las tareas corren en cores con el hyperthread idle.

Libavif avifenc: Este caso resulta particular puesto que con NEST se consigue una degradación del rendimiento en todas las configuraciones de hardware, llegando a ser del 22% en el 6130 de 4 sockets. Aquí, con CFS-schedutil, las tareas empiezan en un socket y poco a poco van migrando a otro, lo que permite turbo-frecuencias medias. Sin embargo, con NEST-schedutil, las tareas quedan ejecutando en el socket inicial, funcionando en la turbo frecuencia más baja.

Other applications

Hackbench and schbench: Este benchmark consiste en pares de procesos que intercambian mensajes. La mayor parte de su tiempo de ejecución reside en la planificación (96% con CFS). Con NEST se obtiene una pérdida de rendimiento notable. De 22,5 segs con CFS a 76 o incluso más de 380 con NEST. Esto se debe al número de instrucciones que fallan en la cache (De 30M con CFS a 49M con NEST o incluso 319M). NEST añade bastante código para hacer la selección de core, y este podría estar más optimizado. Con schbench no se pueden sacar conclusiones ya que a veces dan los mismos resultados, o a veces muy diferentes (mejor CFS o mejor NEST).

Server tests: Se realizan tests sobre diferentes plataformas de servidores web (Node.js, perl, php, etc) y los resultados obtenidos varían en función de uno u otro.

Multiple concurrent applications: Hasta ahora las pruebas han consistido en una única aplicación, por lo que se van a testear varias a la vez. De forma individual, NEST obtiene mejores de rendimiento para zstd (en diferentes modalidades) y Libgav1 Chimera, pero si corremos estas aplicaciones en paralelo, también se observan mejoras que van desde un 3% hasta un 48%, en comparación con CFS multiescenario.

Mono-socket machines: NEST está ideado para ser usado principalmente en grandes servidores, pero también ha sido probado en máquinas más modestas (Intel Xeon 5220 y AMD Ryzen 5 PRO 4650G).

Con el 5220, se obtienen speedups similares para los tests de configuración, mientras que para el resto de benchmarks también se obtienen mejoras, pero no tan acrecentadas como con las grandes máquinas de Intel. Para el NAS benchmark se obtienen exactamente los mismos resultados con CFS y con NEST.

Con el 4650G, los speedups obtenidos para los test de configuración son bastante más altos, tanto empleando performance como schedutil de power governors. Para el resto de benchmarks se obtienen mejoras de en torno al 25%. Al igual que antes, para el NAS benchmark se obtienen exactamente los mismos resultados con CFS y con NEST.

**Related Work**

Una parte que me ha llamado la atención es que a pesar de que NEST concentra toda la computación en un único socket, el hardware no se pone en el último estado de reposo, ya que así se sigue facilitando el acceso a la memoria de esos sockets. Basándose en esto, Nitu et al proponen separar la alimentación de los diferentes sockets, así como de la memoria, para que en grandes centros de datos se pueda ahorrar energía considerablemente manteniendo apagadas aquellas partes de cómputo que no se están utilizando, pero manteniendo accesible el acceso a la memoria.

Otra visión de la asignación de una tarea a un core la da Solaris, favoreciendo la localidad de la memoria en vez del aumento de frecuencia. A día de hoy los servidores pueden modificar la frecuencia de cada core de forma independiente por lo que lo visto en este paper no es suficiente para conseguir el máximo rendimiento.

Relativo a la seguridad, otros papers proponen asignaciones de cores diferentes para que tareas sospechosas no sean ejecutadas en hyperthread, pudiendo provocar así ataques de canal lateral.

**Conclusiones**

NEST muestra un aumento de rendimiento sustancial en aplicaciones donde el número y set de tareas cambia frecuentemente, y mantiene el rendimiento de CFS en aplicaciones que solo requieren una tarea, así como aplicaciones que usan todos los cores de la máquina. La frecuencia del procesador tiene un gran impacto en el rendimiento, permitiendo a las tareas terminar antes. NEST es fácilmente portable a otras versiones de Linux ya que realmente solo es un bloque de código situado delante de CFS.

**Puntos positivos**

Aclaraciones realizadas al finalizar cada uno de los apartados (cores de un mismo die, comparten la cache de último nivel)

Exhaustivos tests

**Puntos negativos**

Densidad de las gráficas

Errores en un documento que se supone ultra revisado

**Preguntas**

**1. Al comienzo de la exposición has comentado que los procesadores son SMT, pero ¿qué es SMT?**

SMT son las siglas de Simultaneous Multithreading, que es una técnica de ejecución empleada a nivel del hardware, con la que un mismo core puede estar ejecutando flujos de ejecución, valga la redundancia, de distinas aplicaciones de manera simultánea, y es que la naturaleza, la microarquitectura de los procesadores actuales es superescalar, es decir, que el pipeline de un core no es único, si no que se encuentra replicado dentro del propio core para favorecer el paralelismo de instrucciones.

**2. Cuando explicabas los dominios de CFS, uno de ellos has comentado que eran los cores de un socket haciendo SMP, pero ¿qué es SMP?**

SMP son las siglas de Simultaneous Multiprocessing, que es una técnica de ejecución a nivel hardware que permite que dos o más cores realicen trabajo paralelo sobre datos compartidos en memoria.

**3. También en la introducción, comentas que la reutilización de cores es especialmente beneficiosa cuando se tiene un número reducido de tareas concurrentes, pero ¿esto por qué es así?**

Esto es así porque la afinidad al core se mantendrá mejor, es decir, con pocas tareas y cortas, será mayor la probabilidad de que cuando una tarea recupere su core, encuentre en la jerarquía de caches los datos que dejó en su última ejecución, además de que cuando vuelva al core sobre el que fue planificada con anterioridad, encontrará que este se encuentra funcionando a una frecuencia de operación elevada.

**4. ¿Siempre resulta mejor utilizar NEST frente a CFS?**

Como hemos visto, este nuevo planificador no siempre resulta mejor, puesto que el rendimiento ofrecido dependerá de la naturaleza de la aplicación que vaya a ejecutar. Al principio se comentó que, cuando el set de tareas tiene una sola, o tantas tareas como cores, el rendimiento obtenido tanto por CFS como por NEST es similar, mientras que, si existen muchas tareas concurrentes, y su duración es reducida, NEST consigue mejores resultados al reducir la dispersión de los threads a lo largo de los cores.

**5. ¿Desde el punto de vista de la seguridad, la adición de líneas de código no podría suponer un problema?**

Realmente, esto sí que podría suponer un problema, ya que es sabido que, por cada 1000 líneas de código, los desarrolladores introducen entre un 10% y un 15% de errores. Pero para poder analizar esto habría que hacer un análisis exhaustivo de la calidad del código y ver que tan explotable serían las vulnerabilidades encontradas.

**6. ¿Consideras que el hardware empleado en las pruebas es adecuado?**

Para tratarse de un paper publicado el año pasado, el hardware empleado parece un tanto desactualizado, puesto que se trabaja con arquitecturas de Intel lanzadas entre los años 2014 y 2019, y en los 3 años que transcurrieron desde el lanzamiento de la última generación hasta el momento de publicación del paper, ha habido tres generaciones de procesadores que no han sido puestas a prueba, y cuyo estudio podría quizá resultar más relevante.

Además, el hardware mencionado en el paper tiene como número máximo de cores 160 (threads), lo que a nuestro modo de ver también se queda algo corto. En la actualidad existen procesadores como los AMD threadripper que ya cuentan por si solos con 96 threads, y si en el documento se recalca que NEST es beneficioso para grandes centros de datos con incluso miles de cores, se deberían haber hecho tests de una mayor magnitud.