UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Máster en Ingeniería de Computadores y Redes

Trabajo Fin de Máster

Estudio de prestaciones de cargas de latencia crítica en sistemas SMT

Autor: Dezhen Wu

Directores: *Josué Feliu, Salvador Petit, Julio Sahuquillo*

Curso 2020/2021

Resumen

La computación en la nube (cloud computing) ofrece servicios de computación bajo demanda a través de una red (habitualmente internet). Es un servicio ampliamente utilizado en la actualidad y, por tanto, existe una gran cantidad de trabajo centrado en el análisis y la mejora de prestaciones de este tipo de servicios. El presente proyecto se centra en cargas de latencia crítica, que son aquellas que deben garantizar una latencia máxima dentro de un umbral para evitar que los usuarios sufran una degradación de prestaciones, cuantificada en términos de calidad de servicio. Aplicaciones como los servicios de búsqueda, el reconocimiento de texto o imágenes y la consulta de bases de datos son aplicaciones de latencia crítica típicas. El presente proyecto propone analizar las prestaciones de este tipo de cargas cuando se ejecutan en un procesador con soporte para la ejecución simultanea de hilos (SMT) utilizando la herramienta perf para determinar las estructuras del procesador que principalmente limitan sus prestaciones.

Palabras clave: TailBench, SMT, latencia crítica, computación en la nube, perf.

Abstract

Cloud computing offers computing services on demand over a network (usually the internet). It is a widely used service today and, therefore, there is a large amount of work focused on the analysis and improvement of the benefits of this type of service. This project focuses on critical latency loads, which are those that must guarantee maximum latency within a threshold to prevent users from suffering a performance degradation, quantified in terms of quality of service. Applications such as search services, text or image recognition, and database queries are typical latency-critical applications. This project proposes to analyze the performance of this type of workloads when running on a simultaneous multithreading (SMT) processor using the perf tool to determine the processor structures that mainly limit their performance.

Keywords: TailBench, SMT, latency-critical, cloud computing, perf.

Tabla de contenidos

1. Introducción al Cloud Computing: problemática, sistemas y cargas

1.1. Cloud Computing

En la última década hemos ampliado nuestro vocabulario relacionado con los temas informáticos con la inclusión de términos como “bigdata”, “nube”, o “computación en la nube”. Los cambios que trae a nuestras vidas esta nueva evolución se han arraigado profundamente en nuestras vidas. Las aplicaciones móviles con las que navegamos diariamente y la visita a los sitios web son básicamente inseparables del poderoso servicio de soporte detrás de la “computación en la nube”. Como muchos sitios web de compras y redes sociales, cambian nuestras vidas.

Según la definición oficial del NIST, “la nube es un modelo de computación que se utiliza para permitir el acceso a la red ubicuo, conveniente y bajo demanda a recursos informáticos configurables (como redes, servidores, almacenamiento, aplicaciones y servicios) a los que se puede acceder rápidamente. Este modelo se puede implementar y lanzar con una mínima interacción de administración o proveedor de servicios. El modelo de nube consta de cinco características básicas (servicio bajo demanda, amplio acceso de red, *pool* de recursos, elasticidad y medición de servicios), tres modelos de servicio (software como servicio o SaaS, plataforma como servicio o PaaS e infraestructura como un servicio o IaaS) y cuatro modelos de implementación (nube privada, nube comunitaria, nube pública y nube híbrida) [1].

Desde que se propuso la computación en la nube en 2006, ha pasado por la etapa de formación, la etapa de desarrollo y la etapa de aplicación. La última década ha sido una década de rápidos avances en la computación en la nube. El mercado mundial de computación en la nube ha crecido en gran medida. Los gobiernos de todo el mundo han formulado una estrategia de "la nube primero". La tecnología de la computación en la nube ha seguido madurando y las aplicaciones de computación en la nube han evolucionado desde la industria de Internet hasta finanzas y la penetración de industrias tradicionales como la industria y el tratamiento médico se ha acelerado [2]. La computación en la nube proporciona servicios informáticos. Empresas multinacionales como Huawei, Amazon, Google, o Microsoft ofrecen servicios incluyendo, entre otros, servidores, almacenamiento, bases de datos, redes, software, análisis e inteligencia. El futuro de computación en la nube es un gran cambio con respecto a los servicios tradicionales.

¿Por qué es tan importante la computación en la nube? Las razones son las siguientes:

- Mediante el uso de la computación en la nube, los clientes (personas, instituciones o empresas) pueden ahorrar el coste de comprar hardware y software, y construir sus propios centros de datos en la nube.
- La computación en la nube puede escalar de manera flexible, lo que significa poder disponer de la cantidad adecuada de servicios de IT.

Estos servicios de computación en la nube se ejecutan en una red global de centros de datos, y estos centros de datos se actualizan periódicamente con la última generación de hardware y software para brindar servicios eficientes. Esto tiene varios beneficios para los usuarios, incluida la reducción de la latencia de la red. La nube también proporciona la mejor solución de seguridad porque proporciona un amplio conjunto de políticas, tecnologías y controles que pueden mejorar la seguridad en su conjunto. Los servicios de computación en la nube brindan a las empresas flexibilidad laboral. Si el alojamiento y la infraestructura de IT dependen de fuerzas externas, los usuarios pueden tener más tiempo para invertir en otros aspectos del negocio, lo que afectará directamente a los resultados de su negocio. Con los servicios en la nube, también se puede obtener fácilmente ancho de banda adicional sin tener que realizar actualizaciones complejas a la infraestructura de IT, que también es muy costosa.

Con estas ventajas, el enorme potencial de la computación en la nube en el futuro se vuelve aún más evidente. La computación en nube y la tecnología detrás de ella tienen muchas oportunidades y capacidades potenciales. En el futuro, puede proporcionar a los clientes una amplia gama de trabajos, servicios, plataformas, aplicaciones, etc.

La computación en la nube avanza a pasos agigantados y es de esperar que el avance se acreciente de manera considerable en la próxima década. Este hecho es gracias a distintos avances. Primero, con el avance de la nueva infraestructura, la computación en la nube acelera el proceso de aplicación y logra un rápido desarrollo en campos como Internet, finanzas, transporte, logística y educación. En segundo lugar, en el contexto de la economía digital global, la computación en la nube se ha convertido en una opción inevitable para la transformación digital de las empresas, y el proceso de las empresas que pasan a la nube se acelerará aún más. En tercer lugar, la aparición de la pandemia del coronavirus ha acelerado la implementación de servicios SaaS como el teletrabajo y la educación online, y ha promovido el rápido desarrollo de la industria de la computación en nube.

1.2. Calidad de servicio, cargas de latencia crítica

Sin embargo, también existen algunos problemas en el proceso de desarrollo. Para las aplicaciones de Internet, la calidad del servicio es un indicador clave de las aplicaciones de Internet. El tiempo de respuesta rápido no solo predice una buena experiencia de usuario, sino que también se convierte en un factor importante para que las empresas de Internet satisfagan y retengan usuarios. Una de las claves son los ingresos de las empresas de Internet. Los estudios han demostrado que si aumenta el tiempo de respuesta del servicio, los ingresos de la empresa disminuirán. Por ejemplo, Amazon descubrió que cada aumento de 100 ms en el tiempo de descarga de amazon.com conlleva una disminución del 1% en las ventas; cuando el tiempo de respuesta del

servicio aumenta de 0,4 segundos a 0,9 segundos, los ingresos por publicidad de Google disminuyen en un 20%. La respuesta de servicio del motor de búsqueda de Microsoft Bing cuando el tiempo aumentó de 50 ms a 2000 ms, la satisfacción del usuario se redujo en un 3,8% y los ingresos por usuario para la empresa se redujeron en un 4,3% [3].

Hoy en día, las aplicaciones de latencia crítica son cada vez más comunes, y estas aplicaciones forman la estructura de servicios interactivos en línea a gran escala. La latencia de cola en lugar de la latencia media es el indicador de rendimiento clave para estas aplicaciones. Estas aplicaciones requieren niveles estrictos de calidad de servicio (QoS) para cumplir con las expectativas del usuario. Por ejemplo, una búsqueda en la web debe completarse en menos de un segundo, de lo contrario, el usuario puede darse por vencido y marcharse. Estudios anteriores han demostrado que los retrasos marginales de QoS (de cientos de milisegundos) afectan en gran medida la experiencia del usuario y los ingresos publicitarios. En particular, es importante cumplir con la latencia de cola (tail latency) de QoS. En general, para su estudio se considera el percentil 95 o 99 de la distribución de *tail latency* [4].

La latencia de algunas aplicaciones es del orden de milisegundos. Por ejemplo, el percentil 99 de la latencia de cola del tiempo de acceso a un nodo hoja de búsqueda web. La necesidad de una latencia de cola baja trae nuevos desafíos y oportunidades para los diseñadores de sistemas, porque muchas tecnologías de hardware y software en los sistemas actuales buscan mejorar el rendimiento promedio a largo plazo, pero no ayudan ni perjudican el rendimiento en el peor de los casos a corto plazo.

Los servicios interactivos en línea a gran escala (como la búsqueda web) deben extraerse a través de conjuntos de datos masivos para satisfacer todas las solicitudes. Estos conjuntos de datos se distribuyen entre cientos o miles de nodos y se almacenan en DRAM o Flash para garantizar una respuesta rápida. Estas cargas de trabajo están diseñadas en una configuración de múltiples niveles y gran distribución, donde el nodo raíz recibe las solicitudes de los usuarios y las traslada a los nodos hoja para su procesamiento. Miles de nodos hoja pueden cooperar para satisfacer cada solicitud de usuario, y el retraso percibido por el usuario está determinado por los pocos nodos más lentos, porque el nodo raíz debe esperar los resultados de la mayoría o de todos los nodos hoja para producir una respuesta final. Por lo tanto, para garantizar un tiempo de espera aceptable de un extremo a otro, la latencia de cola de los nodos hoja (por ejemplo, percentil 95 o 99) debe ser pequeña (por ej., unos pocos milisegundos), y entre los nodos debe ser consistente [5].

1.3. Evolución: procesadores SMT

Al mismo tiempo, en el rápido desarrollo de la computación en la nube, las prestaciones del procesador continúan mejorando constantemente. No hay duda de que la mejora continua del rendimiento del procesador es un fuerte apoyo para el desarrollo y progreso de la computación en la nube.

Los procesadores de altas prestaciones utilizados en los servidores cloud mejoran sus prestaciones incrementando tanto la potencia de cómputo como el subsistema de memoria.

- La potencia de cómputo la incrementan aumentando el número de núcleos y con núcleos que dan soporte a múltiples hilos al mismo tiempo (*simultaneous multithreading* o SMT). En general la mayoría de los procesadores Intel de altas prestaciones dan soporte a dos threads.
- Las prestaciones del subsistema de memoria se incrementan, mediante caches de último nivel (LLC) de varias decenas de MB, varios controladores de memoria integrados en el mismo chip que el procesador y múltiples canales por controlador. De esta manera el procesador aumenta el paralelismo a nivel de thread (*thread level parallelism* o TLP) y el paralelismo en el acceso a memoria (*memory level parallelims* o MLP), lo que le permite satisfacer más peticiones por unidad de tiempo.

1.4. Objetivos del TFM

Este TFM persigue evaluar cargas de latencia crítica en sistemas reales, y cuantificar sus prestaciones, realizando especial énfasis en la latencia de cola. El estudio se realizará en un procesador CPU Intel (R) Xeon (R) E5-2620 v3.

Para conseguir este objetivo general, nos planteamos los siguientes subobjetivos:

- Instalación de la suite TailBench. Se trata de la suite de referencia compuesta por ocho aplicaciones de latencia crítica. Se trata de aplicaciones cliente servidor, donde el cliente realiza una serie de accesos por segundo (QPS o *queries per second*) parametrizable. El servidor se puede encontrar en la misma máquina o en otra distinta. Para este proyecto, consideraremos que se encuentra en la misma máquina.
- Estudio de la latencia de cola en función del número de peticiones por segundo (QPS) realizadas en la aplicación.
- Impacto de SMT o Hyperthreading en la latencia de cola variando el QPS. El estudio considerará hasta 2 hilos de servidor y comparará las diferencias en las prestaciones de ejecutar 1 ó 2 hilos en el mismo núcleo SMT.
- Análisis de prestaciones. Las prestaciones se analizarán a través de indicadores como la tasa de aciertos de la caché de primer nivel, el número de fallos de cache por cada mil instrucciones (MPKI) en todos los niveles de la cache y las causa de *stalls* del procesador. El objetivo es conocer las razones que limitan el rendimiento a partir de los resultados del análisis, que desempeñarán un papel importante en la mejora del rendimiento del procesador y la reducción de la latencia de cola de las aplicaciones críticas.

2. Antecedentes

2.1. Procesadores SMT

La investigación actual sobre tecnología de procesadores y arquitectura de computadores se basa principalmente en la demanda de un mayor rendimiento. En este caso, es bien sabido que la mejora del rendimiento aportada por mejorar el sistema de memoria es limitada. Los beneficios más significativos se pueden lograr aumentando el grado de paralelismo en la ejecución.

El procesador puede extraer principalmente tres tipos de paralelismo: paralelismo a nivel de instrucción (ILP), paralelismo a nivel de memoria (MLP) y paralelismo a nivel de hilo (TLP). Las arquitecturas superescalares modernas están diseñadas para extraer ILP y MLP en un único proceso. Sin embargo, el ILP y MLP que puede extraerse de un único hilo de ejecución está muy limitado por las dependencias entre instrucciones. Por ello, el TLP surge como una opción muy interesante para conseguir mejorar el paralelismo tanto a nivel de instrucción como a nivel de memoria, combinando la ejecución de varios hilos en núcleos distintos (procesadores multinúcleo) o dentro del mismo núcleo (procesadores multihilo).

Existen diversos paradigmas multihilo. Los principales son el multihilo de grano grueso (*coarse-grain multithreading*), el multihilo de grano fino (*fine-grain multithreading*) y el multihilo simultáneo (*simultaneous multithreading* o SMT)[6]. Los procesadores multihilo de grano grueso intercambian el hilo en ejecución en eventos de larga latencia como accesos a memoria principal [7] [8] procesadores multihilo de grano fino intercambian el hilo en ejecución en cada ciclo o intervalos de muy corta latencia [9] [10]. Esto les permite conseguir una mejor utilización de los recursos del procesador y unas mayores prestaciones. En consecuencia, SMT es el paradigma multihilo más utilizado en los procesadores comerciales actuales.

La figura 1 muestra los *issue slots* utilizados y desperdiciados para un procesador que permite lanzar a ejecución 4 instrucciones por ciclo. Los procesadores multihilo de grano grueso y fino tienen como objetivo eliminar el desperdicio vertical de *issue slots*. Para ello, ante un evento de alta latencia (multihilo de grano grueso) o cada ciclo (multihilo de grano fino), estos paradigmas multihilo cambian el hilo en ejecución. Sin embargo, no reducen el desperdicio horizontal de *issue slots*. Si un hilo no es capaz de lanzar cuatro instrucciones a ejecución, los *issue slots* que no pueda utilizar se desperdician. Los procesadores SMT, en cambio, reducen tanto el desperdicio vertical como el horizontal ya que permiten ejecutar múltiples instrucciones de varios hilos en el mismo ciclo.

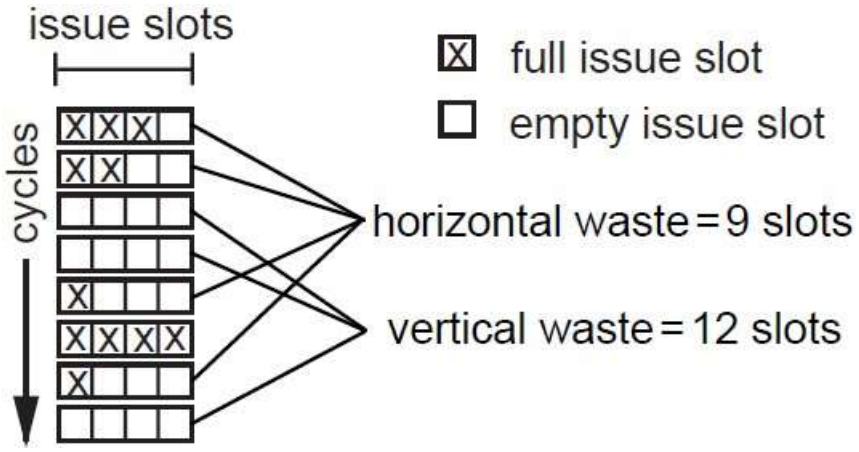

Figura 1: Desperdicio de Issue slots.

Aunque los procesadores multi-hilo han existido desde la década de 1950, el primer microprocesador comercial importante desarrollado con SMT fue el Alpha 21464 (EV8). El microprocesador fue desarrollado por DEC en colaboración con Dean Tullsen de la Universidad de California, San Diego y Susan Eggers y Henry Levy de la Universidad de Washington. El microprocesador nunca se lanzó porque la serie Alpha de microprocesadores se suspendió poco antes de que HP adquiriera Compaq y Compaq adquiriera DEC. No obstante, los resultados de Dean Tullsen también se han utilizado para desarrollar versiones Hyper-threading (o HTT) de microprocesadores Intel Pentium 4, como "Northwood" y "Prescott".

Intel Pentium 4 es el primer procesador de escritorio que implementa SMT. Intel llama a su implementación Hyper-Threading y proporciona un motor SMT básico de doble hilo. Intel afirma que es un 30% más rápido que el Pentium 4 no SMT manteniendo el resto de los parámetros. Como ya hemos comentado, el objetivo del SMT es aumentar significativamente la utilización del procesador para hacer frente a una latencia de memoria prolongada y a un paralelismo disponible limitado por hilo. Tal y como se observó en los trabajos de investigación [11] a mejora de rendimiento observable depende en gran medida de las aplicaciones en ejecución por lo que la planificación de los procesos adecuada tiene un gran impacto en las prestaciones finales de un sistema SMT [12].

En las elecciones de diseño de microarquitectura tomadas para la implementación del procesador SMT, Intel consideró tres objetivos clave. Uno de los objetivos es minimizar el incremento del área para implementar la tecnología SMT. De hecho, la implementación de la tecnología Hyper-Threading solamente ha aumentado el tamaño del área de la oblea en menos de 5% y proporciona mejoras de prestaciones mucho más significativas. El segundo objetivo es garantizar que cuando un procesador lógico deja de funcionar, el otro procesador lógico puede seguir adelante. El tercer objetivo es permitir que cuando el procesador ejecuta solo un hilo software, éste se ejecute a la misma velocidad a la que lo haría en un procesador no SMT [13].

La arquitectura SMT implementada por Intel se refinó con la introducción de la arquitectura Nehalem y desde entonces ha sufrido cambios pequeños hasta llegar a los procesadores SMT comerciales fabricados por Intel. La compartición de las diferentes estructuras del núcleo se realiza de cuatro formas distintas:

- Estructuras replicadas. Cada hilo tiene su propia copia de las estructuras replicadas. Esto significa que cuando el procesador ejecuta un único hilo, las estructuras del otro hilo quedan sin utilizar. Las estructuras involucradas en mantener el estado arquitectónico de cada hilo son el ejemplo más claro de estructuras replicadas.
- Estructuras particionadas. Estas estructuras se dividen en dos mitades cuando el procesador ejecuta dos hilos y cada hilo puede acceder a la mitad de la estructura. El particionado es interesante porque tiene un coste muy pequeño en área y complejidad. Entre las estructuras particionadas en los procesadores SMT de Intel están el *load buffer*, el store buffer o el *reorder buffer*.
- Estructuras compartidas. Estas estructuras se comparten dinámicamente por los hilos en ejecución. Cada hilo puede utilizar tantas entradas en estas estructuras como desee. Entre las estructuras compartidas se encuentran las caches o las estaciones de reserva.
- Estructuras *unaware*. Algunas estructuras como las unidades funcionales no son conscientes de si las instrucciones que ejecutan pertenecen a un hilo o a dos hilos diferentes.

Es interesante comentar que los núcleos SMT pueden implementarse sin problemas en los procesadores multinúcleo o CMP. Cada núcleo dentro de un CMP puede disponer de soporte para la ejecución simultánea de hilos. Los procesadores multinúcleo de Intel implementan núcleos SMT con soporte para la ejecución simultanea de dos hilos. La propuesta de IBM es mucho más ambiciosa y en los procesadores IBM POWER8 [14][15].

Al observar las tendencias de software actuales, puede encontrar que las aplicaciones de servidor están compuestas por múltiples hilos o procesos que se pueden ejecutar en paralelo. Por ejemplo, el procesamiento de transacciones en línea y los servicios web tienen una gran cantidad de hilos de software, que se pueden ejecutar simultáneamente para mejorar el rendimiento. Además, estos hilos también pueden provenir de diferentes aplicaciones. Al agregar más procesadores o procesadores con soporte SMT, las aplicaciones pueden potencialmente aumentar significativamente el rendimiento al ejecutar múltiples hilos en múltiples procesadores simultáneamente.

2.2. Cargas de latencia crítica y la suite TailBench

Como se mencionó anteriormente, existen algunas tareas interactivas o tareas con estrictos requisitos de latencia en el centro de datos, como motores de búsqueda, reconocimiento de imágenes, bases de datos, etc., que son muy sensibles a la latencia. El tiempo de espera de estas tareas de latencia crítica (tareas LC) solo puede estar dentro de un cierto rango de tiempo, como decenas de milisegundos, de lo contrario, afectará la experiencia del usuario del servicio. Para aquellas tareas o servicios orientados al usuario que son críticos con la latencia, la latencia de cola (es decir, la latencia de las peticiones más lentas) es un indicador importante para medir la calidad del servicio y la experiencia del usuario. Existe un cierto intervalo de latencia desde el envío de la solicitud hasta la recepción de la respuesta. Cuando la latencia supera este

tiempo (denominado objetivo de nivel de servicio (SLO) en algunos estudios), la calidad de servicio es inaceptable.

Por lo general, debido a diversas razones, algunas de ellas aleatorias, es difícil asegurar que todas las solicitudes cumplan el SLO. Por ello, se puede aceptar que un pequeño porcentaje de las solicitudes tengan una latencia superior al SLO. En consecuencia, la métrica utilizada para evaluar la latencia de un servicio suele ser la latencia de cola, es decir, la latencia de las peticiones con mayor latencia (en lugar de utilizar la latencia media). Generalmente, los estudios utilizan latencias de cola del percentil 95 y 99 para evaluar si se cumple o no el SLO.

La suite de referencia con aplicaciones de latencia crítica es la suite TailBench [5]. A diferencia de suites anteriores centradas en una computación más tradicional, como por ejemplo, la suite SPEC CPU2006 que se centra en el rendimiento y productividad del procesador más la memoria cache y principal, la suite TailBench se centra en la latencia de cola en el sistema completo. Y además de un conjunto de aplicaciones de latencia crítica incluye una serie de scripts y código adicional para facilitar la configuración de los experimentos y el análisis de la latencia de cola.

La suite TailBench incluye un conjunto de ocho aplicaciones típicas de latencia crítica e integra todas las cargas de trabajo en un entorno común (*harness*) para implementar un método sólido y estadísticamente razonable para realizar los experimentos. El *harness* se encarga de generar las peticiones en función del número de peticiones por segundo requerido siguiendo una distribución que sea representativa en entornos cloud (por ejemplo, una distribución Zipfian), de medir el tiempo de servicio y el tiempo de espera en cola de las aplicaciones, y por último, de recopilar y agregar las latencias para proporcionar la latencia de cola del percentil solicitado. Además, TailBench proporciona tres configuraciones de red distintas para lanzar las aplicaciones utilizando un mismo nodo como cliente y servidor o nodos diferenciados para cliente y servidor. En sus estudios indican que la configuración de un único nodo permite realizar los experimentos de forma satisfactoria midiendo la misma latencia para la mayoría de las aplicaciones a la vez que simplifica el coste y el tiempo necesario para desarrollar los experimentos. Por tanto, en este trabajo usaremos la configuración de un único nodo.

A continuación, describimos las ocho aplicaciones que forman la suite.

Img-dnn es una aplicación de reconocimiento de escritura a mano basada en OpenCV. El reconocimiento de escritura a mano es un ejemplo de una categoría más amplia de aplicaciones de reconocimiento de imágenes, que ahora se utilizan ampliamente en el reconocimiento óptico de caracteres, búsquedas basadas en imágenes (como Google Goggles), etiquetado automático de imágenes y otras aplicaciones en línea. Img-dnn utiliza un codificador automático basado en redes neuronales profundas y regresión softmax para reconocer caracteres escritos a mano. El benchmark utiliza muestras seleccionadas al azar de la base de datos MNIST para su ejecución.

Masstree es un almacén de clave-valor (*key-value store*) en memoria rápido y escalable escrito en C++. El almacén de clave-valor en memoria sirve como backend de almacenamiento de datos para varios servicios. Los almacenes de clave-valor

ocupan grandes cantidades de datos, que se dividen en fragmentos residentes en la memoria distribuidos entre cientos de servidores. Cada solicitud de usuario suele incluir decenas o cientos de solicitudes de almacenamiento de valores clave. Por lo tanto, estas aplicaciones tienen requisitos de latencia muy cortos, por ejemplo, alrededor de 100 µs. Además, esta aplicación fue elegido para la suite porque está altamente optimizado para utilizar de manera efectiva la jerarquía de memoria de procesadores multinúcleo modernos. Se utiliza masstree con una versión modificada de Yahoo Cloud Serving Benchmark , que tiene un 50% de peticiones get y un 50% de peticiones put.

Moses es el sistema de traducción automática estadística más avanzado escrito en C++. El sistema es la base de los servicios de traducción en línea como Google Translate y una parte importante de las interfaces basadas en voz como Apple Siri. El benchmark utiliza el decodificador basado en frases incluido en moses; moses también admite la decodificación basada en árboles. Para ejecutar moses se utilizan fragmentos de diálogo seleccionados al azar del corpus inglés-español de opensubtitles.org.

Shore es una base de datos transaccional. A diferencia de silo, es una base de datos en disco y es muy diferente en la forma de almacenar y acceder a los datos. Se utiliza TPC-C para ejecutar shore. Para evitar cuellos de botella en la E/S del disco, la base de datos y los registros se almacenan en unidades de estado sólido.

Silo es una base de datos de transacciones de memoria rápida. Silo tiene como objetivo escalar en los procesadores multinúcleo modernos, evitando puntos de contención concentrados y utilizando eficazmente la jerarquía de memoria. Las bases de datos como silo se utilizan ampliamente en los sistemas de procesamiento de transacciones en línea (OLTP). El benchmark utiliza TPC-C (benchmark OLTP estándar de la industria) para ejecutar silo.

Specjbb es la prueba comparativa de middleware Java estándar de la industria. El middleware de Java se utiliza ampliamente en los servicios empresariales y, por lo general, debe cumplir con estrictas restricciones de latencia. Specjbb simula un sistema de 3 niveles, que es exclusivo de muchas aplicaciones Java del lado del servidor. El sistema modelado es una empresa mayorista que maneja diferentes tipos de solicitudes de clientes (por ejemplo, procesamiento de pagos y entregas). El benchmark utiliza HotSpot v1.8 para ejecutar specjbb.

Sphinx es un sistema de reconocimiento de voz preciso escrito en C ++. El sistema de reconocimiento de voz es una parte importante de las interfaces y aplicaciones basadas en voz (como Apple Siri, Google Now e IBM Speech to Text). El reconocimiento de voz es una actividad computacionalmente intensiva que implica la poda probabilística de grandes árboles de búsqueda. Sphinx utiliza sofisticados modelos acústicos, fonéticos y modelos de lenguaje para mejorar la eficiencia y precisión. Para ejecutar sphinx se utilizan pronunciaciones seleccionadas al azar de la base de datos alfanumérica CMU AN4.

Xapian es un motor de búsqueda de código abierto escrito en C++, ampliamente utilizado en sitios web populares (como Debian Wiki) y marcos de software (como Catalyst). Los motores de búsqueda en línea procesan índices de datos del orden de petabytes, que se dividen en fragmentos distribuidos en miles de nodos hoja. La mayor

parte del procesamiento se realiza en los nodos hoja y cada nodo busca su parte de índice de forma independiente. Xapian está configurado para representar el nodo hoja. El índice de búsqueda se construye a partir de un volcado de la versión en inglés de Wikipedia en julio de 2013. Las condiciones de consulta se seleccionan aleatoriamente de acuerdo con la distribución de Zipfian, que ha demostrado simular bien la distribución de consultas de búsqueda en línea.

3. Trabajo relacionado

La computación en la nube tiene una alta escalabilidad, flexibilidad y rentabilidad, y puede satisfacer las necesidades informáticas emergentes. Para proporcionar recursos informáticos en la nube de manera eficaz, los administradores de sistemas deben tener la capacidad de caracterizar y predecir cargas de trabajo en máquinas virtuales (VM). A medida que más y más aplicaciones en la nube han pasado de trabajos de procesamiento por lotes a servicios con estrictos requisitos de latencia en los últimos años, la investigación sobre las características de las cargas de trabajo en la nube se ha vuelto particularmente importante. Existen investigaciones anteriores en términos de nivel de virtualización (por ejemplo, VM, contenedor o sin virtualización), carga de trabajo de investigación (por ejemplo, HPC, nube, etc.) y fuentes de interferencia de destino (por ejemplo, LLC, ancho de banda de memoria, E/S, etc.).

En distintos trabajos [16][17][18][19][20] se ha llevado a cabo una extensa investigación sobre la caracterización y predicción de la carga de trabajo. Sin embargo, esta serie de trabajos se centra en la creación de modelos matemáticos que se pueden utilizar para representar cargas de trabajo típicas de servidores web, plataformas o redes informáticas de alto rendimiento. Al igual que en [21][22][23][24], estos estudios caracterizan la carga de trabajo en el entorno de la nube. Sin embargo, estos estudios están relacionados con la comprensión estadística y la reproducción de tareas informáticas (como las tareas de MapReduce) programadas en la nube. La gestión de la capacidad y la implementación de máquinas virtuales suelen utilizar algunas técnicas de previsión y modelado de cargas de trabajo [25][26][27][28][29]. Estos métodos dependen de la información estadística de cada serie de tiempo de carga de trabajo para predecir los requisitos de recursos futuros. Algunos trabajos existentes como [30][31] solo se enfocan en la interferencia en LLC y/o ancho de banda de la memoria, y no consideran otros recursos compartidos importantes (como la red o el disco) en el entorno de nube. Además, estos trabajos no consideraron la virtualización, por lo que los resultados pueden no ser aplicables a entornos virtualizados.

El trabajo anterior resolvió principalmente la interferencia de tres maneras. Su método principal es simplemente prohibir que los servicios interactivos compartan recursos con otras aplicaciones para evitar interferencias [32][33][34][35]. Esto puede preservar la QoS de la aplicación, pero reducirá la utilización de recursos del sistema. El segundo método es evitar la programación conjunta de aplicaciones que puedan interferir entre sí [36][37][38]. Esto limita las opciones de aplicaciones que se pueden programar conjuntamente. El tercer método se centra en eliminar completamente la interferencia mediante el uso de técnicas de aislamiento a nivel de sistema operativo y de hardware para asignar recursos entre servicios programados conjuntamente [39][40][41][42]. Este método protege la calidad del servicio de los servicios. Desafortunadamente, este método está actualmente limitado a un máximo de un servicio interactivo por host físico y está programado conjuntamente con uno o más trabajos BE (*best-effort*). Alternativamente, si se programan conjuntamente varias

aplicaciones interactivas en un host físico, su carga se reducirá considerablemente, lo que resultará en una utilización insuficiente [43].

En [44] se llevó a cabo una investigación detallada en dos aplicaciones representativas de latencia crítica. El enfoque de la investigación es considerar el impacto de diferentes factores en el rendimiento (como plataforma experimental, carga de entrada, virtualización, etc.). En [45] se obtienen trazas de datos de entornos reales de nube privada y se presentan características de carga de trabajo, pero el enfoque principal de la investigación se centra identificar patrones de carga de trabajo comunes relacionados con la utilización de CPU de aplicaciones comerciales de red. En [46], se describe el impacto de la interferencia en los recursos compartidos a diferentes niveles de carga, sin embargo, solo se consideran tres cargas de trabajo de la latencia crítica y no se analiza la interferencia en el disco. En contraste, [47] consideró la interferencia del disco y estudió seis aplicaciones críticas de latencia. Sin embargo, ambos tipos de trabajo utilizan virtualización LXC [48] (es decir, contenedores Linux) en lugar de máquinas virtuales ligeras.

En este trabajo nos centraremos en cómo afecta la utilización de núcleos con soporte SMT a las prestaciones de las cargas de latencia crítica, algo que se no ha abordado en ninguno de los trabajos previos.

4. Entorno experimental

En esta sección se presentan los componentes clave del entorno experimental, incluyendo el sistema hardware, los contadores de prestaciones, y el sistema operativo y herramientas software instaladas.

4.1. Hardware

El entorno experimental está basado en un sistema con procesador Intel (R) Xeon (R) CPU E5-2620 v3, el cuál es un procesador multinúcleo de 64 bits fabricado en tecnología 32 nanómetros. Este procesador está orientado a servidores y típicamente se instala en sistemas con soporte de hasta dos sockets de procesador. La figura 2 muestra dos procesadores Intel® Xeon® E5-2400 instalados en un sistema con 2 sockets. Los sockets se conectan con un enlace punto a punto Intel® QuickPath Interconnect (Intel® QPI). El sistema cuenta soporte para 24 interfaces PCI Express 3.0 y 4 PCI Express 2.0. También integra un controlador de memoria (IMC) y entrada/salida (IIO).

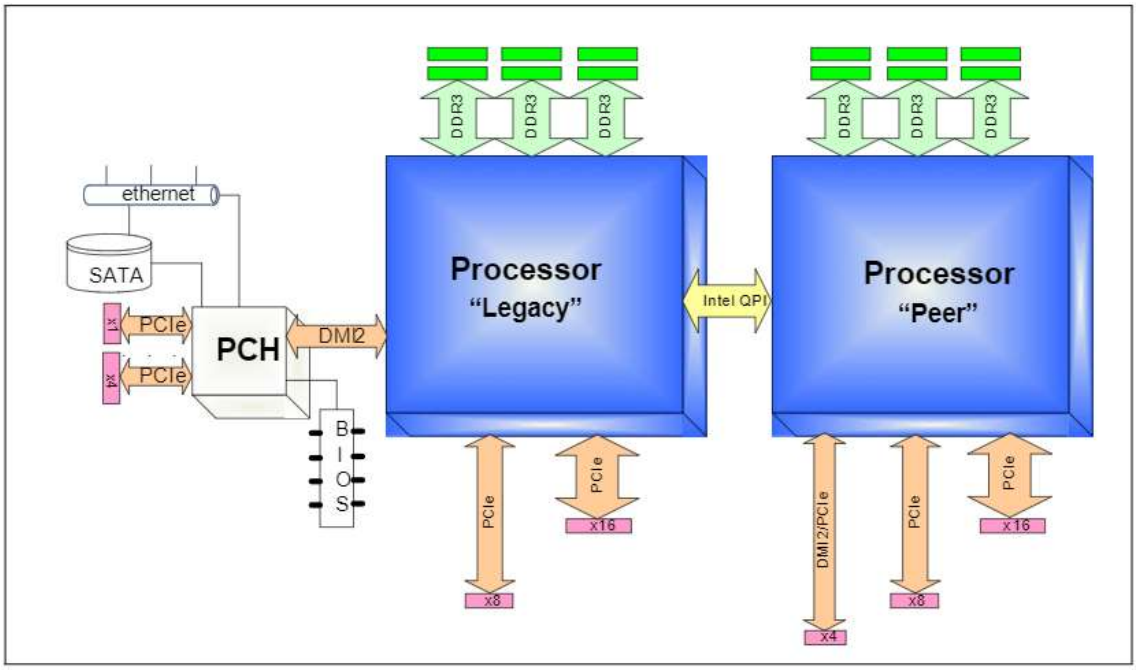

Figura 2: Procesadores Intel® Xeon® de la familia E5-2400 en una plataforma con 2 sockets.

Las características del procesador son:

- Ocho núcleos (cores) físicos.

- Cada núcleo soporta dos hilos (tecnología Intel® Hyper-Threading o SMT), lo que permite ejecutar hasta 16 hilos por procesador.
- Direccionamiento físico de 46 bits y direccionamiento virtual de 48 bits.
- Caches de instrucciones (L1I) y datos (L1D) de 32 KB privadas para cada núcleo.
- Caché de segundo nivel (L2) de 256 KB privada para cada núcleo.
- La caché de último nivel (LLC) tiene una capacidad de 20MB, y es compartida entre los 8 núcleos de un socket.

4.2. Contadores de prestaciones

En las últimas décadas, los sistemas computacionales incluyen procesadores cada vez más potentes para mejorar el rendimiento. Sin embargo, no siempre es posible confiar en estas mejoras para proporcionar unas mayores prestaciones. Más aún, con los procesadores multinúcleo y los sistemas con acceso a memoria no uniforme (NUMA) es posible experimentar pérdidas de rendimiento debido a problemas de contención o sobrecarga en el acceso a recursos compartidos como la memoria principal o la cache LLC. Por lo tanto, los desarrolladores deben evaluar cuidadosamente su software a través de un análisis basado en mediciones para encontrar problemas de rendimiento y sus causas.

El análisis del rendimiento del software es uno de los temas principales de la informática tanto en la academia como en la industria. Este análisis permite revelar problemas de prestaciones del software en todas las etapas de desarrollo. Para ello, el software bajo estudio se ejecuta bajo una carga de trabajo específica y se miden los indicadores para asegurar que los últimos cambios en el programa no causan ninguna regresión en las prestaciones.

Generalmente, se usan tres métodos de análisis de rendimiento: a) basados en muestreo, b) instrumentación a nivel de instrucción (paso a paso), y c) basados en contadores hardware de prestaciones. El primer método tiene poca sobrecarga, pero proporciona resultados relativamente imprecisos. En comparación, el segundo método proporciona información mucho más precisa, pero su sobrecarga es bastante elevada, lo que impide su utilización en sistemas funcionando a pleno rendimiento. Finalmente, los métodos basados en contadores hardware ofrecen un seguimiento no intrusivo y detallado del comportamiento del sistema, lo que es una de las razones principales de su utilización en este proyecto.

Los contadores hardware proporcionan información directa a nivel de la microarquitectura del procesador, incluyendo, entre otros, los ciclos de ejecución, los *stalls*, y los accesos a los diversos niveles de la jerarquía de memoria. En los sistemas basados en arquitectura x86-64, los contadores se acceden a través de registros especiales denominados MSR. Existen dos tipos de información accesible mediante estos registros, la información de conteo (es decir, el contador), que cuantifica el número de eventos de un determinado tipo que ocurren durante la ejecución y la

información de control, que debe configurarse para seleccionar qué eventos del hardware se monitorizan.

En el sistema operativo Linux, el acceso a los contadores hardware se estructura en tres niveles software:

- **Controlador de *Performance Monitoring Unit (PMU)***. Este nivel es dependiente de la arquitectura del procesador y normalmente su implementación requiere el uso de instrucciones en ensamblador para acceder a los registros MSR.
- **Interfaz del Kernel**. La interfaz del kernel utiliza el controlador de la PMU para proporcionar, a través de llamadas al sistema, acceso a los contadores del hardware por parte del espacio de usuario. El desarrollo de la interfaz se realiza mediante propuestas de parches al kernel de Linux, que debe recompilarse una vez parcheado para soportar la interfaz.
- **Herramientas del nivel de usuario**. Estas herramientas aprovechan la interfaz del kernel para ofrecer acceso a los contadores hardware a usuarios y programadores. Las herramientas permiten analizar las aplicaciones basándose en los recuentos de eventos proporcionados por los contadores hardware. Otras funciones básicas de estas herramientas incluyen: análisis de rendimiento, recuentos de eventos, gráficos de llamadas al sistema e informes de rendimiento avanzados.

La tabla 1 ilustra la estructura jerárquica de los distintos niveles de acceso a los contadores hardware en el sistema operativo Linux [49].

<table>
<tr><td rowspan="2">Machine Independent</td><td>User Level Tools</td><td>Linux Perf, PAPI, perfmon2, OProfile</td><td>User Space</td></tr>
<tr><td>Kernel Interfaces</td><td>Perf_event, pefmon2, perfctr</td><td rowspan="2">Kernel Space</td></tr>
<tr><td>Machine Dependent</td><td>PMU Drivers</td><td>X86, armv7, i686, powerpc, Sparc</td></tr>
</table>

Tabla 1: Jerarquía del software para acceder a los contadores hardware en Linux.

4.3. Sistema operativo y herramientas software

El sistema ejecuta Ubuntu 18.04.4 LTS, una distribución del sistema operativo GNU/Linux que incorpora el núcleo Linux 5.3. Este sistema operativo soporta diversas herramientas (p.e. perf, ftrace, strace, top, tcpdump, etc.) para ayudar a los desarrolladores a encontrar cuellos de botella en el rendimiento del software. Cada una

de estas herramientas está diseñada para monitorizar partes específicas del sistema. En esta sección se describen las herramientas taskset y perf, utilizadas en este trabajo para la asignación de procesos a núcleos y la monitorización de los contadores hardware de prestaciones.

4.3.1. Taskset

Una característica de los sistemas operativos que a menudo se explota para mejorar el rendimiento de aplicaciones críticas en procesadores multinúcleo es la capacidad de establecer la denominada afinidad de un proceso a uno o varios núcleos. Esta característica permite definir en qué núcleo o núcleos se ejecuta cada proceso con el objetivo de limitar la interferencia entre procesos y proveerles de suficiente potencia computacional.

Vincular un programa a núcleos específicos puede resultar beneficioso en varios escenarios. Por ejemplo, una aplicación con una carga de trabajo muy afectada por la cache disponible puede compartir el mismo núcleo sólo con aplicaciones con un uso menos intensivo de cache, lo que puede mejorar su rendimiento. En otros casos, por ejemplo, cuando dos hilos de una aplicación paralela se comunican a de forma intensiva a través de la memoria compartida, anclar ambos hilos a núcleos del mismo dominio NUMA acelera el rendimiento global de la aplicación paralela [50].

En este trabajo, ejecutamos el cliente y el servidor en el mismo sistema. Por lo tanto, para reducir la interferencia entre cliente y servidor, las ejecutamos en núcleos distintos utilizando la herramienta taskset.

La herramienta taskset forma parte del paquete «util-linux». La mayor parte de las distribuciones Linux vienen con este paquete preinstalado. En caso de que taskset no esté disponible, es posible instalarlo en un sistema Ubuntu de la siguiente manera:

```
$ sudo apt-get install util-linux
```

El taskset se utiliza para establecer u obtener la afinidad de núcleo de un proceso en ejecución dado su pid, o para lanzar un nuevo comando con una afinidad determinada. La afinidad se representa como una máscara de bits, con el bit de menor peso correspondiente al primer núcleo lógico y el bit mayor peso correspondiente al último núcleo. Las máscaras se pueden especificar en hexadecimal o como una lista de núcleos con la opción --cpu-list. Por ejemplo, la máscara 0x00000001 referencia exclusivamente al núcleo lógico #0, mientras que la máscara 0x00000003 referencia a los núcleos #0 y #1 ($3 = 0011_2$). Finalmente, la máscara 0xFFFFFFFF representa a todos los núcleos del sistema.

El siguiente cuadro describe el comando taskset y sus diversas opciones [51]:

Sinopsis
taskset *[options] mask command [arg]...*
taskset *[options]* **-p** *[mask] pid*

Opciones

-p, --pid

Operar en un PID existente y no iniciar una nueva tarea

-c, --cpu-list

Especificar una lista numérica de procesadores en lugar de una máscara de bits. La lista puede contener varios elementos, separados por comas y rangos. Por ejemplo, **0,5,7,9-11**.

-h, --help

Mostrar información de uso y salir

-V, --version

Generar información de versión y salir

Uso

El comportamiento predeterminado es ejecutar un nuevo comando con una máscara de afinidad determinada:

taskset *mask command [argumentos]*

También puede recuperar la afinidad de CPU de una tarea existente:

taskset -p *pid*

O configúrelo:

taskset -p *mask pid*

Permisos

Un usuario debe poseer **CAP_SYS_NICE** para cambiar la afinidad de CPU de un proceso. Cualquier usuario puede recuperar la máscara de afinidad.

4.3.2. Perf

La optimización del rendimiento de una aplicación generalmente incluye dos etapas: análisis y optimización. El objetivo del análisis de rendimiento es encontrar los cuellos de botella y sus causas. El objetivo de la optimización es mejorar las prestaciones a partir del análisis de rendimiento mediante la modificación del código de la aplicación, las opciones de compilación y la distribución de los recursos hardware del sistema[52].

En la etapa de análisis de rendimiento, es necesario recurrir a herramientas como perf. Perf es una herramienta de análisis de rendimiento disponible para el núcleo de Linux desde su versión 2.6 en adelante. Perf es una de las herramientas de *profiling* de contadores de prestaciones más utilizadas en Linux. Al principio se centró exclusivamente en los contadores hardware de prestaciones pero hoy abarca todo tipo de eventos relacionados con el rendimiento. Perf dispone de una interfaz de línea de comandos mediante la cual se puede acceder a la PMU, puntos de control y estadísticas del kernel. La figura 3 muestra los diversos módulos que componen perf tanto en espacio de usuario como de kernel.

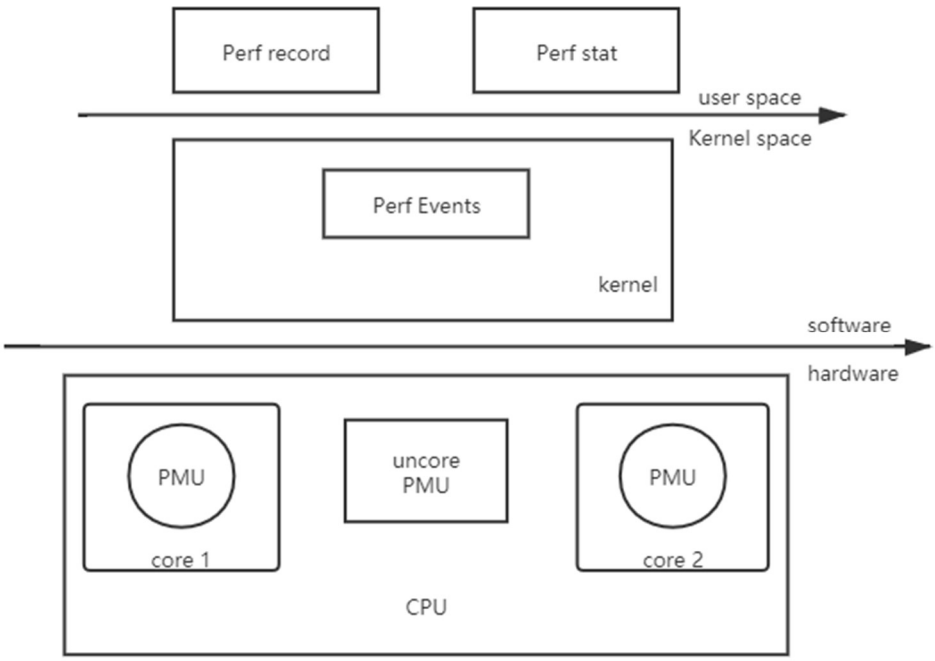

Figura 3: Arquitectura de perf.

Perf es capaz de no sólo analizar los problemas de rendimiento hilos de aplicación específicos, sino también de analizar el rendimiento del kernel. Además, también puede analizar el código de la aplicación y el kernel al mismo tiempo. En comparación con otras aplicaciones similares como OProfile y GProf, perf tiene la ventaja de estar estrechamente integrado en el kernel de Linux y puede beneficiarse de las características más recientemente incorporadas en este.

Perf puede limitar la medición a un ámbito específico de la ejecución. Por ejemplo, puede realizar muestreos de acuerdo con la interrupción del reloj del sistema o sólo cuando se llama a determinada función de la aplicación o del núcleo [53]. Esta potencia permite al desarrollador de aplicaciones o del kernel averiguar dónde debe centrar sus esfuerzs de optimización y evaluar el uso de los recursos hardware como el número de accesos a cada nivel de cache, el número de fallos de cada nivel, stalls del procesador, accesos a memoria principal, etc. Además, puede muestrear eventos del sistema operativo como llamadas al sistema, cambios de contexto y migración de tareas [54].

Los eventos que puede muestrear perf se pueden dividir en tres tipos:

- Eventos hardware. Son eventos generado por la PMU del procesador, como un acierto en cache. Cuando se necesita entender el uso que hace el programa de los recursos del hardware, deben monitorizarse estos eventos;
- Eventos software. Son eventos generados por el kernel, como cambios de proceso, número de ciclos de reloj, etc.
- Eventos asociados a puntos de control. Son eventos desencadenados por puntos de control definidos en el kernel Estos puntos de control se utilizan para determinar los detalles del comportamiento del kernel durante la ejecución de la aplicación, como los tiempos de asignación de recursos como podría ser espacio de memoria dinámica.

La instalación de perf es muy simple, desde el directorio raíz del código fuente del kernel, se ingresa en el directorio *tools/perf* y se introducen los siguientes comandos:

```
~/tools/perf $ make

~/tools/perf $ make install
```

La herramienta Perf proporciona un amplio conjunto de opciones para recopilar y analizar datos de seguimiento y rendimiento. El uso de la línea de comandos es similar al de git. Es decir, la interfaz presenta un conjunto de subcomandos, que se describen a continuación:

usage: perf [--version] [--help] COMMAND [ARGS]

Los subcomandos perf más utilizados son:

annotate	Read perf.data (created by perf record) and display annotated code
archive	Create archive with object files with build-ids found in perf.data file
bench	General framework for benchmark suites
buildid-cache	Manage <tt>build-id</tt> cache.
buildid-list	List the buildids in a perf.data file
diff	Read two perf.data files and display the differential profile
inject	Filter to augment the events stream with additional information
kmem	Tool to trace/measure kernel memory(slab) properties
kvm	Tool to trace/measure kvm guest os
list	List all symbolic event types
lock	Analyze lock events
probe	Define new dynamic tracepoints
record	Run a command and record its profile into perf.data
report	Read perf.data (created by perf record) and display the profile
sched	Tool to trace/measure scheduler properties (latencies)
script	Read perf.data (created by perf record) and display trace output
stat	Run a command and gather performance counter statistics
test	Runs sanity tests.
timechart	Tool to visualize total system behavior during a workload
top	System profiling tool.

Ciertos subcomandos requieren un soporte especial en el kernel y pueden no estar disponibles. Para obtener la lista de opciones para cada subcomando, simplemente se introduce el nombre del subcomando seguido de -h. Así, por ejemplo:

```
perf stat -h

 usage: perf stat [<options>] [<command>]

    -e, --event <event>     event selector. use 'perf list' to list available events
    -i, --no-inherit        child tasks do not inherit counters
    -p, --pid <n>           stat events on existing process id
    -t, --tid <n>           stat events on existing thread id
    -a, --all-cpus          system-wide collection from all CPUs
    -c, --scale             scale/normalize counters
    -v, --verbose           be more verbose (show counter open errors, etc)
    -r, --repeat <n>        repeat command and print average + stddev (max: 100)
    -n, --null              null run - dont start any counters
    -B, --big-num           print large numbers with thousands' separators
```

5. Soporte experimental y metodología

En esta sección se describe la metodología utilizada para realizar los experimentos de este TFM así como el soporte necesario para su ejecución. En primer lugar, se explica cómo instalar la suite TailBench, compuesta por aplicaciones de latencia crítica, objetivo de este estudio. En segundo lugar, se describe la metodología experimental y los scripts que soportan esta metodología. La metodología incluye la configuración de los experimentos y la monitorización de los contadores hardware de prestaciones.

5.1 Instalación y compilación de la suite TailBench

Para instalar la suite TailBench, deben realizarse los siguientes pasos:

1. Descargamos TailBench desde su repositorio en GitHub:

```
$ git clone https://github.com/lyuhao/tailbench
```

2. Se descargan y descomprimen las cargas de entrada para TailBench:

```
$ wget -c http://tailbench.csail.mit.edu/tailbench.inputs.tgz
$ tar xzvf tailbench-v0.9.tgz
```

3. Se instalan los paquetes Ubuntu con las dependencias de TailBench:

```
$ sudo apt-get install openjdk-8-jdk libopencv-dev autoconf ant libtcmalloc-minimal4 swig google-perftools libboost1.58-all-dev bzip2 libnuma-dev libjemalloc-dev libgoogle-perftools-dev libdb5.3 ++ - dev libmysqld- dev libaio-dev uuid-dev libbz2-dev python-numpy python-scipy libgtop2-dev
```

4. Antes de compilar TailBench, es necesario configurar el archivo config.sh siguiendo los siguientes pasos:

```
$ cd tailbench_v2
~ / tailbench_v2 $ nano configs.sh
```

5. Dependiendo del sistema, puede que se necesite configurar las variables JDK_PATH y DATA_ROOT de la siguiente forma:

```
#config.sh
# Set this to point to the top level of the TailBench data directory
DATA_ROOT=/home/dwu/tailbench.inputs

# Set this to point to the top level installation directory of the Java
# Development Kit. Only needed for Specjbb
JDK_PATH=/usr/lib/jvm/java-8-openjdk-amd64

# This location is used by applications to store scratch data during execution.
SCRATCH_DIR=/home/dwu/pathtoscratch
```

Una vez hayamos completado los pasos anteriores, es el momento de compilar la suite. Para evitar errores en el proceso de compilación debido a la falta de algunas librerías, compilaremos e instalaremos las librerías faltantes una por una. En primer lugar, compilaremos el componente *harness*, encargado de planificar la carga y medir las prestaciones obtenidas:

```
$ ./build.sh harness
```

A continuación, las ocho aplicaciones que componen la suite se compilan por separado. Por ejemplo, para compilar silo se debe ejecutar:

```
$ ./build.sh silo
```

5.2 Configuración de los experimentos y metodología

Una vez completada la compilación, para realizar el estudio de prestaciones presentado en este trabajo, se definen tres experimentos. En cada uno de estos experimentos se ejecutan todas las aplicaciones de latencia crítica de la suite TailBench con una configuración diferente para servidor y cliente. En particular, las configuraciones son las siguientes:

- En el primer experimento, configuramos un servidor de un solo hilo y 5 hilos de cliente.
- En el segundo experimento, configuramos el servidor para que utilice dos hilos sin aprovechar el soporte SMT. Es decir, ejecutamos los dos hilos del servidor en diferentes núcleos físicos.
- En el tercer experimento, configuramos el servidor con dos hilos aprovechando el soporte SMT. Es decir, ambos hilos del servidor se ejecutan en el mismo núcleo físico, cada uno de los hilos en uno de los núcleos lógicos.

Para poder definir la distribución de los hilos en cada una de esas configuraciones entre los núcleos físicos y lógicos, primero es necesario obtener la correspondencia entre núcleos lógicos y físicos mediante el siguiente comando:

$ lscpu -e

CPU	NODE	SOCKET	CORE	L1d:L1i:L2:L3	ONLINE	MAXMHZ	MINMHZ
0	0	0	0	0:0:0:0	yes	3200.0000	1200.0000
1	0	0	1	1:1:1:0	yes	3200.0000	1200.0000
2	0	0	2	2:2:2:0	yes	3200.0000	1200.0000
3	0	0	3	3:3:3:0	yes	3200.0000	1200.0000
4	0	0	4	4:4:4:0	yes	3200.0000	1200.0000
5	0	0	5	5:5:5:0	yes	3200.0000	1200.0000
6	0	0	0	0:0:0:0	yes	3200.0000	1200.0000
7	0	0	1	1:1:1:0	yes	3200.0000	1200.0000
8	0	0	2	2:2:2:0	yes	3200.0000	1200.0000
9	0	0	3	3:3:3:0	yes	3200.0000	1200.0000
10	0	0	4	4:4:4:0	yes	3200.0000	1200.0000
11	0	0	5	5:5:5:0	yes	3200.0000	1200.0000

A partir de la salida obtenida se puede observar claramente la correspondencia entre cada núcleo lógico (CPU en la salida) y núcleo físico (CORE). Con esta información, se modifican los scripts run_network.sh de las ocho aplicaciones para cumplir con los requisitos experimentales de cada configuración. Asimismo, este script se modifica para automatizar los experimentos, incluyendo la asignación de cada proceso al núcleo lógico correspondiente y la monitorización de los contadores hardware de prestaciones. Para estas tareas, utilizaremos las herramientas taskset y perf introducidas previamente. Además, el script run_network.sh incluye algunas variables de configuración, las cuales se describen en la tabla 2.

Variable	Descripción
TBENCH_WARMUPREQS	Período de tiempo de calentamiento durante el cual no se realizan mediciones.
TBENCH_MAXREQS	Cantidad de solicitudes que deben ejecutarse durante la medición, excluyendo las solicitudes de preparación.
TBENCH_MINSLEEPNS	Período de tiempo mínimo (en ns) que el programa cliente duerme en el período de inactividad.
TBENCH_QPS	Número promedio de solicitudes durante el período de medición (el número de consultas por segundo). *Harness* utiliza una distribución exponencial para generar el intervalo de tiempo entre solicitudes.
TBENCH_RANDSEED	Semilla del generador de números aleatorios.
TBENCH_CLIENT_THREADS	Cantidad de solicitudes generadas por los hilos del cliente.
TBENCH_SERVER	URL o dirección IP del servidor. El predeterminado es localhost.
TBENCH_SERVER_PORT	Puerto TCP / IP utilizado por el servidor. El valor predeterminado es 8080.

Tabla 2: Variables de script-run_network.sh.

Como ejemplo, se presenta el script correspondiente al benchmark silo:

```
#!/bin/bash
# ops-per-worker is set to a very large value, so that TBENCH_MAXREQS controls
how
# many ops are performed

if [ "$#" -ne 3 ]; then
   echo "Usage: ./run_networked.sh N_CLIENTS TBENCH_CLIENT_THREADS
CLIENT_QPS."
   exit 0
fi

NUM_WAREHOUSES=1
NUM_THREADS=2
N_CLIENTS=1
TBENCH_CLIENT_THREADS=4
QPS=$1
WARMUPREQS=$((N_CLIENTS*QPS*4))
MAXREQS=$((WARMUPREQS*3))

threads=$(echo "${N_CLIENTS}*${TBENCH_CLIENT_THREADS=}" | bc -l)

echo RESULT N_CLIENTS: ${N_CLIENTS} QPS: ${QPS} WARMUPREQS:
${WARMUPREQS} MAXREQS: ${MAXREQS}

TBENCH_MAXREQS=${MAXREQS}
TBENCH_WARMUPREQS=${WARMUPREQS}
TBENCH_NCLIENTS=${N_CLIENTS} taskset -c 0,6 \
perf stat -e cpu-
cycles,instructions,mem_load_uops_retired.l1_hit,mem_load_uops_retired.l1_mi
ss,mem_load_uops_retired.l2_miss,mem_load_uops_retired.l3_miss,\
cycle_activity.stalls_l1d_pending,cycle_activity.stalls_l2_pending,cycle_activity.st
alls_ldm_pending,\
resource_stalls.any,resource_stalls.rob,resource_stalls.rs,resource_stalls.sb,uops_
executed.stall_cycles \
      ./out-perf.masstree/benchmarks/dbtest_server_networked --verbose --bench
\
   tpcc --num-threads ${NUM_THREADS} --scale-factor
${NUM_WAREHOUSES} \
   --retry-aborted-transactions --ops-per-worker 10000000 2>&1 | tee server.sal &

echo $! > server.pid

sleep 5 # Allow server to come up
```

```
for ((i=0; i<${N_CLIENTS}; i++)); do
    TBENCH_QPS=${QPS} TBENCH_MINSLEEPNS=100
TBENCH_CLIENT_THREADS=${TBENCH_CLIENT_THREADS}
TBENCH_ID=${i} \
    taskset -c 2,3,4,5 ./out-perf.masstree/benchmarks/dbtest_client_networked
2>&1 | tee client_${i}.sal &
    echo $! > client_${i}.pid
done

wait $(cat server.pid)

# Clean up
for ((i=0; i<${N_CLIENTS}; i++)); do
    kill -9 $(cat client_${i}.pid) > /dev/null 2>&1
    rm client_${i}.pid
done

kill -9 $(cat server.pid) > /dev/null 2>&1
rm server.pid
```

Nótese que el script tiene como parámetro de entrada ($1) las peticiones por segundo (Queries Per Second o QPS) generadas por el cliente. Variando el QPS se pueden modificar la carga del servidor y estudiar el efecto sobre el rendimiento (latencia de cola del 95%) y contadores hardware de prestaciones. La latencia de cola se obtiene parseando la salida del experimento con el script:

```
$ ../utilities/parselats.py
```

Los eventos monitorizados con contadores hardware de prestaciones, los cuales se obtienen con la herramienta perf, se muestran en la tabla 3.

Evento	**Descripción**
Instructions	Number of instructions executed.
cpu-cycles	Number of processor cycles consumed.
mem_load_uops_retired.l1_miss	Retired load uops misses in L1 cache as data sources.
mem_load_uops_retired.l2_miss	Miss in mid-level (L2) cache. Excludes Unknown data-source.
mem_load_uops_retired.l3_miss	Miss in last-level (L3) cache. Excludes Unknown data-source.

cycle_activity.stalls_l1d_pending	Execution stalls due to L1 data cache misses
cycle_activity.stalls_l2_pending	Execution stalls due to L2 cache misses.
cycle_activity.stalls_ldm_pending	This event counts cycles during which no instructions were executed in the execution stage of the pipeline and there were memory instructions pending (waiting for data).
resource_stalls.any	Resource-related stall cycles.
resource_stalls.rob	Cycles stalled due to re-order buffer full.
resource_stalls.rs	Cycles stalled due to no eligible RS entry available.
resource_stalls.sb	This event counts cycles during which no instructions were allocated because no Store Buffers (SB) were available.
uops_executed.stall_cycles	Counts number of cycles no uops were dispatched to be executed on this thread.

Tabla 3: Eventos monitorizados por los contadores de prestaciones.

Utilizando estos contadores, se definen las métricas de prestaciones, que se explicarán en el siguiente capítulo.

6. Resultados y análisis

Como se mencionó en el capítulo anterior, realizamos tres conjuntos de pruebas bajo tres condiciones diferentes. En este capítulo analizaremos los cambios en las prestaciones de cada aplicación y sus razones a través de los datos recogidos para cada experimento a través de los contadores de prestaciones. Como métrica de prestaciones utilizaremos la latencia de cola del percentil 95 (*95th tail-latency*). Las configuraciones estudiadas son *"1 hilo"*, *"2 hilos - 2 núcleos"* y *"2 hilos - SMT"* que corresponden a la ejecución del servidor con un solo hilo, con 2 hilos asignando cada hilo a núcleos distintos, y con 2 hilos asignados al mimo núcleo SMT.

Este capítulo se divide en tres partes. En la primera parte analizamos la variación de la latencia de cola del percentil 95 variando el QPS para las tres configuraciones. En la segunda parte definimos algunas métricas basadas en los eventos monitorizados con los contadores de prestaciones para analizar las diferencias de prestaciones entre las diversas configuraciones. En la tercera, realizamos este análisis y explicamos los motivos por los que algunas aplicaciones alcanzan las mismas prestaciones cuando se ejecutan en un núcleo SMT que en dos núcleos, mientras que otras aplicaciones alcanzan más prestaciones al usar dos núcleos distintos.

6.1. Latencia de cola del percentil 95

Antes de pasar a presentar los resultados para cada aplicación, podemos avanzar que para las ocho aplicaciones de latencia crítica que ejecutamos, como podíamos esperar, el rendimiento del servidor cuando se ejecuta con dos hilos es siempre mayor que cuando se ejecuta con un único hilo, aunque los dos hilos se ejecuten utilizando un único núcleo SMT. Es decir, la latencia de cola del percentil 95 de la configuración "2 hilos - 2 núcleos" y de la configuración "2 hilos - SMT" es menor que la latencia de cola del percentil 95 de "1 hilo" cuando este último empieza a saturar. Aunque con un número de peticiones por segundo (QPS) bajo la latencia de cola de las tres configuraciones es similar ya que un único hilo es capaz de servir las peticiones lo suficientemente rápido como para que no esperen encoladas, las configuraciones del servidor con 2 hilos consiguen mantener una latencia de cola baja para una carga superior y, por tanto, podemos afirmar que alcanzan un rendimiento mayor.

A continuación, analizaremos en detalle la situación de cada aplicación y centraremos el análisis en la latencia de cola que alcanzan las dos configuraciones del servidor con dos hilos. La figura correspondiente para cada aplicación muestra la latencia de cola del 95% para cada una de las tres configuraciones estudiadas a medida que aumenta la carga del servidor, cuantificada en peticiones por segundo (QPS).

- **Img-dnn**. En primer lugar, la Figura 4 muestra la latencia de cola del percentil 95 para las tres configuraciones de la aplicación img-dnn. Podemos ver que la configuración con 2 hilos en 2 núcleos distintos es la que soporta un QPS más alto antes de que la latencia de cola crezca de forma brusca. Además, incluso para los QPS más bajos, esta configuración proporciona una latencia de cola algo inferior. Las configuraciones de 1 hilo y de 2 hilos en un núcleo SMT alcanzan una latencia de cola similar con un QPS inferior a 1250 peticiones por segundo, aproximadamente. No obstante, mientras que la configuración con un hilo incrementa la latencia de cola de forma drástica en este punto, la configuración con 2 hilos en un núcleo SMT la mantiene más o menos estable hasta las 1750 peticiones por segundo. Por tanto, la configuración con 2 hilos en 2 núcleos alcanza el mayor rendimiento claramente entre las tres configuraciones y, la configuración de 2 hilos en un núcleo SMT mejora las prestaciones de la configuración con un único hilo.

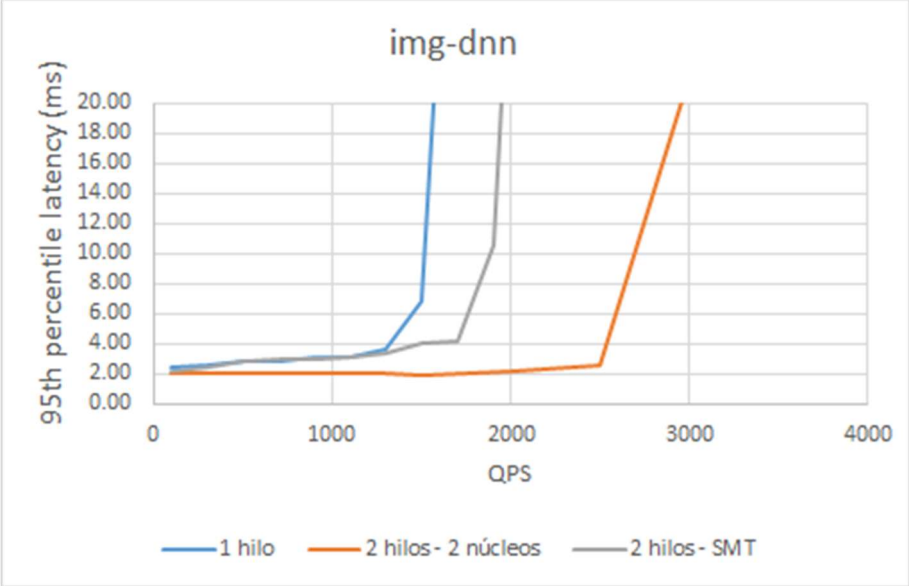

Figura 4: Latencia de cola del percentil 95 para para img-dnn.

- **Masstree**. La figura 5 muestra como varía la latencia de cola del percentil 95 de masstree en las tres configuraciones. La figura muestra que las tres configuraciones alcanzan una latencia de cola muy similar hasta 2500 peticiones por segundo, donde la latencia de cola para la configuración del servidor con un único hilo empieza a crecer significativamente. Las dos configuraciones con dos hilos consiguen mantener la latencia de cola más o menos estable hasta las 4500 peticiones por segundo. La latencia de cola satura en 5000 peticiones por segundo y crece exponencialmente. A pesar de que para este QPS la latencia de cola es algo inferior para la configuración del servidor con 2 hilos en 2 núcleos, la diferencia con la configuración de 2 hilos en un núcleo SMT es muy poca. Podemos concluir, por tanto, que las prestaciones para las dos configuraciones del servidor con 2 hilos son muy similares entre ellas y muy superiores a las que alcanza el servidor con un único hilo.

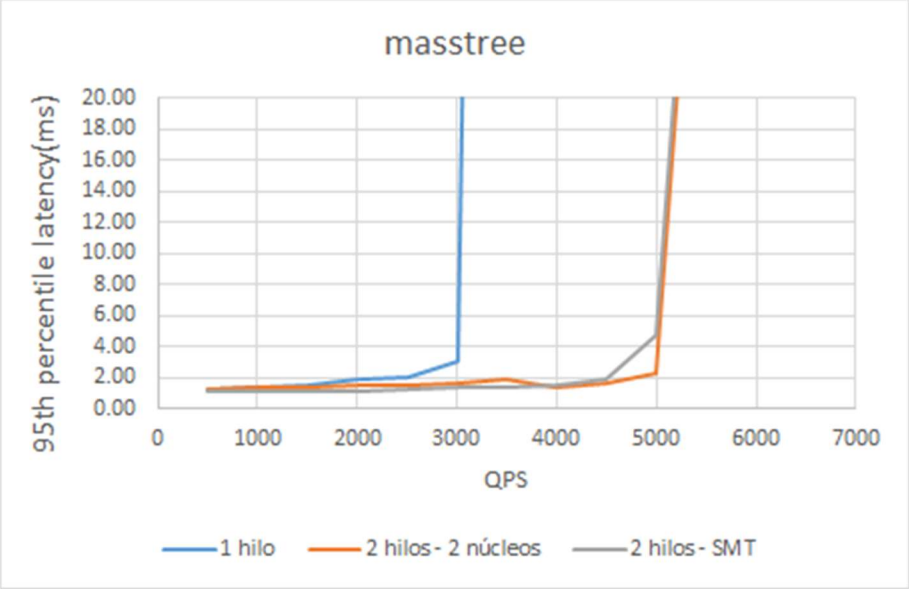

Figura 5: Latencia de cola del percentil 95 para para masstree.

- **Moses**. La figura 6 muestra la latencia de cola del percentil 95 para las tres configuraciones estudiadas al ejecutar la aplicación moses. En esta aplicación las tres configuraciones tienen una latencia similar para 25 peticiones por segundo. A partir de este número de peticiones por segundo, las prestaciones difieren. La latencia crece rápidamente a partir de 50-75 peticiones por segundo para la configuración del servidor con un hilo, a partir de 100 peticiones por segundo para la configuración de 2 hilos en un núcleo SMT y a partir de 125-150 peticiones para la configuración de 2 hilos en dos núcleos. Por tanto, la configuración 2 hilos en 2 núcleos tiene el rendimiento más alto, seguida por la configuración de 2 hilos en un núcleo SMT y, por última, la configuración de 1 hilo.

Figura 6: Latencia de cola del percentil 95 para para moses.

- **Shore**. La figura 7 muestra la latencia de cola del percentil 95 para la aplicación shore. A partir de la figura podemos ver que en la configuración de un hilo, la latencia de cola aumenta de la siguiente forma: a medida que aumenta QPS a partir de 25 peticiones por segundo, la latencia de cola crece progresivamente y a partir de 125 peticiones por segundo satura y crece de forma exponencial. En cambio, la latencia de cola para las dos configuraciones de 2 hilos es casi idéntica entre ellas y se mantiene más o menos estable hasta las 75 peticiones por segundo. A partir de ahí crece de forma más o menos rápida, pero siempre por debajo de la latencia de cola de la configuración de un hilo. Por tanto, para esta aplicación las prestaciones son la mismas al ejecutar los dos hilos en un núcleo SMT que cuando se utilizan dos núcleos distintos para ejecutarlos.

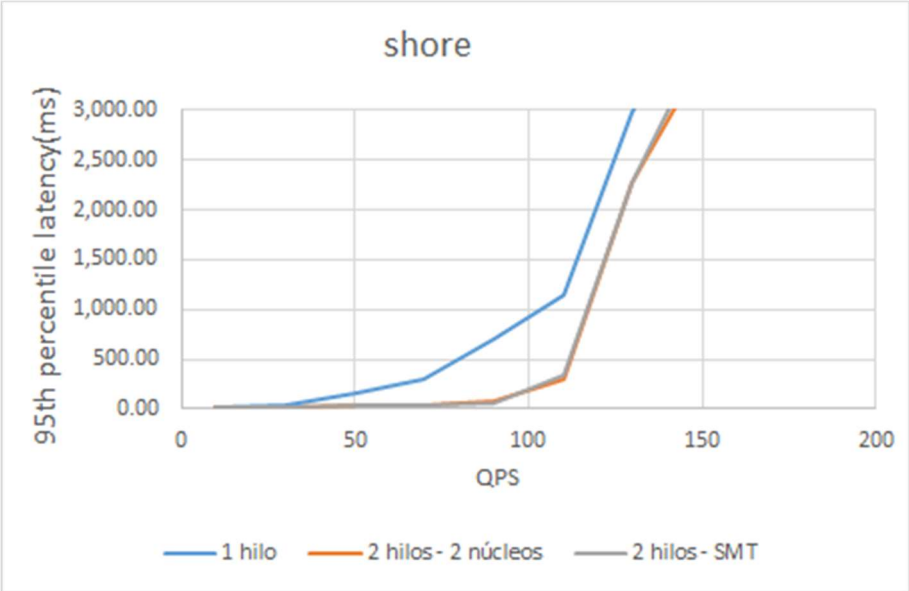

Figura 7: Latencia de cola del percentil 95 para para shore.

- **Silo**. Como muestra la Figura 8, la latencia de Silo tiene una tendencia similar en las tres configuraciones: la latencia se mantiene más o menos estable por debajo de 1ms hasta que el servidor satura, empieza a no poder responder las peticiones al mismo ritmo que llegan y la latencia sube exponencialmente. Cuando se ejecuta el servidor con un único hilo, la latencia se mantiene más o menos estable por debajo de 1ms hasta un QPS de 2500 peticiones por segundo. Cuando se utilizan dos hilos, la latencia se mantiene estable hasta las 4000 peticiones por segundo. La observación importante es que el servidor de Silo alcanza las mismas prestaciones cuando los dos hilos se ejecutan en el mismo núcleo SMT que cuando lo hacen en dos núcleos distintos. El comportamiento es similar al descrito para shore y sugiere que el núcleo no es el principal cuello de botella y que la utilización de los recursos del núcleo que realiza un único hilo es relativamente baja.

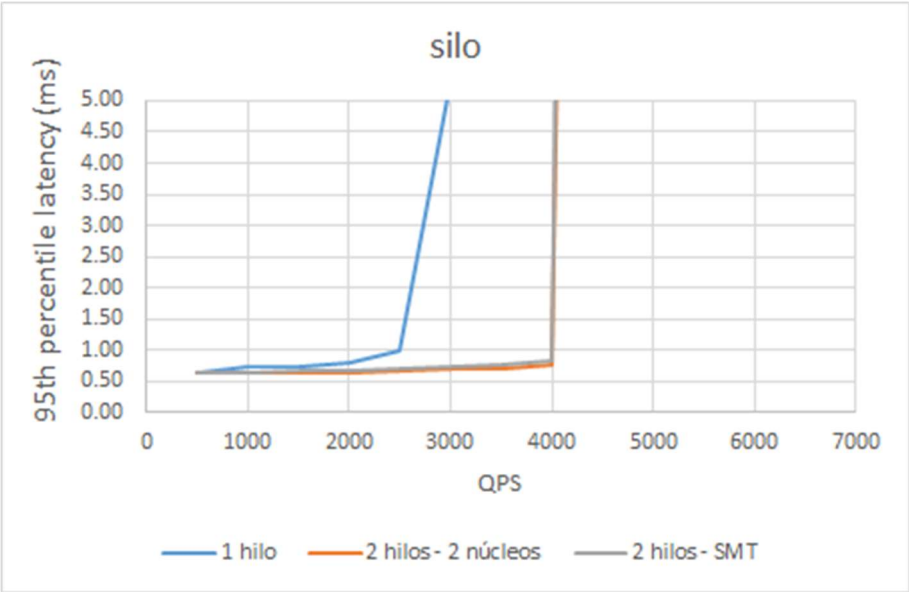

Figura 8: Latencia de cola del percentil 95 para para silo.

- **Specjbb**. La figura 9 muestra la variación de la latencia de cola del percentil 95 al ejecutar la aplicación specjbb en las tres configuraciones. Como se ve, en la configuración de 1 hilo, existe un valor crítico de QPS a partir del cual la latencia se dispara (3000 peticiones por segundo para esta configuración). Por debajo de este QPS la latencia de cola se mantiene estable y no excede 1,5 ms para ninguna de las tres configuraciones. Las dos configuraciones con dos hilos mantienen la latencia estable hasta las 3500 peticiones por segundo, proporcionando un rendimiento algo superior al de la configuración de un hilo. En este caso ejecutar 2 hilos en un núcleo SMT mejora ligeramente las prestaciones de ejecutar un único hilo y ejecutar los dos hilos en dos núcleos distintos no proporciona ninguna mejora en comparación con ejecutarlos en el mismo núcleo SMT.

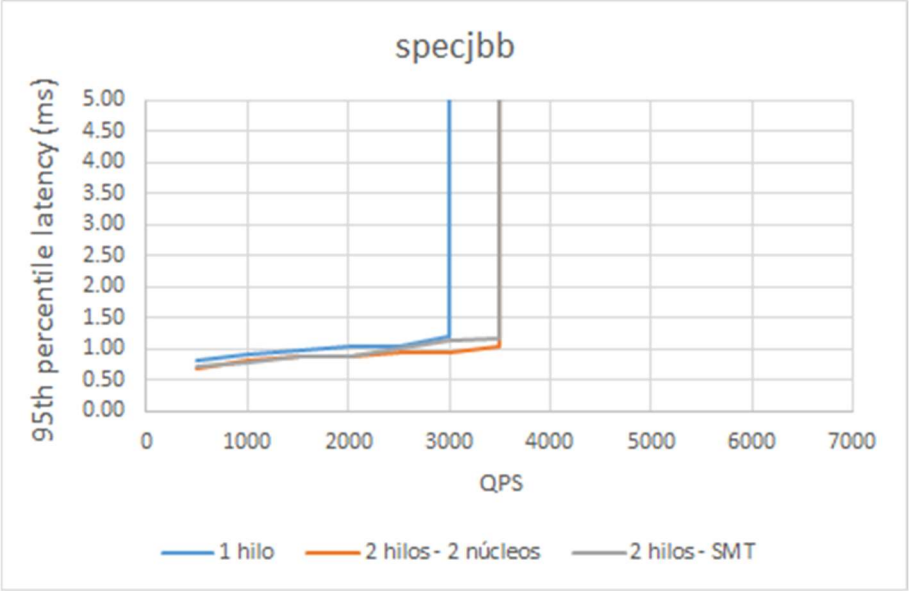

Figura 9: Latencia de cola del percentil 95 para para specjbb.

- **Sphinx**. La Figura 10 muestra la latencia de cola del percentil 95 de sphinx en las tres configuraciones estudiadas. Como se puede observar, sphinx tienen un comportamiento distinto al del resto de aplicaciones estudiadas ya que la latencia crece de forma más o menos continúa desde prácticamente 0.1 peticiones por segundo. Además, las peticiones de sphinx tienen un tiempo de procesamiento muy superior a las del resto de aplicaciones y su latencia es mayor. Las tres configuraciones tienen un rendimiento distinto en sphinx siendo la configuración de un hilo la que ofrece una peor latencia de cola, seguida de la configuración de 2 hilos en un núcleo SMT y, por último, la configuración de 2 hilos en 2 núcleos distintos, que es la que ofrece una latencia de cola más baja. Al ejecutar sphinx, es importante utilizar núcleos distintos y ejecutar dos hilos en un núcleo SMT para tener el mayor rendimiento.

Figura 10: Latencia de cola del percentil 95 para para sphinx.

- **Xapian**. Por último, la Figura 11 muestra la variación de la latencia de cola del percentil 95 para xapian. La latencia de cola de xapian se mantienen más o menos estable por debajo de 10ms para las tres configuraciones hasta que las prestaciones saturan y la latencia de cola aumenta repentinamente. Esto ocurre con 1000 peticiones por segundo para la configuración de un hilo, 1400 par la configuración de 2 hilos en un núcleo SMT y 2000 par la configuración de 2 hilos en dos núcleos diferentes. Por tanto, para esta aplicación ejecutar los 2 hilos en dos núcleos diferentes sí que proporciona una mejora del rendimiento muy importante en comparación con ejecutarlos en un núcleo SMT.

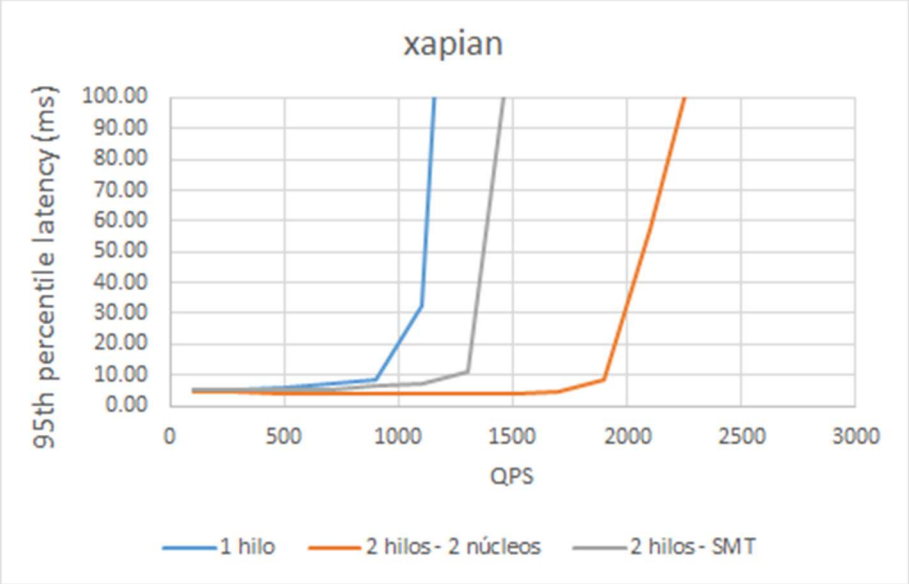

Figura 11:Latencia de cola del percentil 95 para para xapian.

En resumen, a partir de los resultados mostrados para todas las aplicaciones, podemos concluir que, como podíamos esperar, todas las aplicaciones tienen mejores prestaciones cuando se ejecutan con dos hilos que cuando se ejecutan con un único hilo. Sin embargo, su comportamiento de función de si los hilos se ejecutan en el mismo núcleo SMT o si lo hacen en dos núcleos distintos es diferente. Un grupo de aplicaciones (img-dnn, moses, sphinx y xapian) alcanzan unas mayores prestaciones cuando los dos hilos se ejecutan en dos núcleos distintos. En cambio, para otro grupo de aplicaciones (masstree, shore, silo y specjbb) ejecutar los dos hilos en dos núcleos distintos no ofrece ninguna mejora de prestaciones en comparación con ejecutarlos en el mismo núcleo SMT.

Esta observación es muy importante de cara a optimizar el rendimiento de los servidores cloud. Ejecutar dos hilos del servidor en un núcleo SMT permite liberar un núcleo para ejecutar otras cargas y, como hemos visto, en algunas aplicaciones esto se consigue sin sufrir ninguna perdida de prestaciones en comparación con ejecutar el servidor utilizando dos núcleos diferentes.

6.2. Métricas analizadas

En la sección anterior, hemos analizado como varía la latencia de cola de las aplicaciones en las tres configuraciones estudiadas observando dos tendencias claramente diferenciadas. En algunas aplicaciones, la configuración del servidor de dos hilos utilizando un núcleo SMT alcanza las mismas prestaciones que cuando los dos hilos se ejecutan en núcleos distintos. En cambio, en otras aplicaciones, utilizar dos núcleos distintos proporciona mejoras de prestaciones muy significativas. En esta sección vamos a definir diferentes métricas que utilizaremos para analizar las

diferencias de prestaciones y estimar que aplicaciones pueden ejecutarse utilizando dos hilos en un núcleo SMT sin perder prestaciones.

La Tabla 4 muestra las métricas utilizadas junto con una descripción y la fórmula utilizada para calcularlas. Como se mencionó anteriormente, usamos la herramienta perf para monitorizar diferentes eventos hardware y a partir de estos eventos calculamos las métricas de prestaciones que utilizaremos en el análisis.

Métricas	Descripción	Fórmula
IPC	Instrucciones por ciclo	instructions /cpu-cycles
TAL1	Tasa de acierto en la cache L1	mem_load_uops_retired.l1_hit / (mem_load_uops_retired.l1_hit+mem_load_uops_retired.l1_miss)
MPKI L1	Fallos en la cache L1 por cada mil instrucciones	mem_load_uops_retired.l1_miss / (instructions/1000)
MPKI L2	Fallos en la cache L2 por cada mil instrucciones	mem_load_uops_retired.l2_miss / (instructions/1000)
MPKI L3	Fallos en la cache L3 por cada mil instrucciones	mem_load_uops_retired.l3_miss / (instructions/1000)
MAPUS	Accesos a memoria por microsegundo	mem_load_uops_retired.l3_miss * 3200MHz/cpu-cycles
Dispatch Stalls	*Dispatch stalls* por ciclo (incluyo los provocados por recursos y por otras causas)	uops_executed.stall_cycles / cpu-cycles
% Stalls RES	Porcentaje de *dispatch stalls* causados por recursos (incluye ROB, RS, SB y otros recursos).	resource_stalls.any/ uops_executed.stall_cycles
% Stalls ROB	Porcentaje de *dispatch stalls* causados por el ROB	resource_stalls.rob / uops_executed.stall_cycles
% Stalls RS	Porcentaje de *dispatch stalls* causados por las estaciones de reserva	resource_stalls.rs / uops_executed.stall_cycles
% Stalls SB	Porcentaje de *dispatch stalls* causados por el SB	resource_stalls.sb / uops_executed.stall_cycles

Tabla 4: Métricas utilizadas para analizar las prestaciones.

6.3. Análisis de las prestaciones

De la evaluación de la latencia de cola del percentil 95 podemos observar que, en algunas aplicaciones, tener 2 hilos en diferentes núcleos tiene un mejor rendimiento

que tener 2 hilos en el mismo núcleo SMT. Este es el caso de img-dnn, moses, sphinx y xapian. En otros casos, en cambio, las diferencias de prestaciones entre ejecutar los hilos en núcleos distintos o en el mismo núcleo son muy pequeñas o inexistentes. A partir de esta observación, la Table 5 clasifica las aplicaciones en dos grupos: *SMT-friendly* y *No SMT-friendly*. En el primer grupo tenemos las aplicaciones que son favorables a usar SMT y tienen prestaciones similares que cuando se usan núcleos distintos para ejecutar los hilos. En el segundo grupo están las aplicaciones que sí que obtienen mayores prestaciones cuando los hilos se ejecutan en núcleos distintos.

SMT-friendly	No SMT-friendly
Masstree	Img-dnn
Shore	Moses
Silo	Sphinx
specjbb	Xapian

Tabla 5: Clasificación de las aplicaciones en SMT-friendly y No SMT-friendly.

La evaluación de las aplicaciones muestra que agregar hilos en el lado del servidor reduce efectivamente la latencia de cola de las aplicaciones de latencia crítica. Sin embargo, queda por analizar más a fondo el impacto de la tecnología SMT en las aplicaciones de latencia críticas. A continuación, evaluaremos las métricas definidas en la sección anterior para analizar las causas por las que algunas aplicaciones son *SMT-friendly* y otras no lo son. Esté análisis podrá servir de guía para determinar la mejor forma de asignar los hilos de las aplicaciones en los núcleos en servidores cloud en función de si una aplicación se beneficiará o no de la tecnología SMT.

Para analizar las características de las aplicaciones, nos centraremos en su ejecución con un único hilo y utilizaremos como referencia el valor de QPS más alto antes de que el servidor se sature y la latencia de cola del percentil 95 aumente exponencialmente. La Tabla 6 muestra el QPS utilizado para cada aplicación en este análisis.

Aplicación	Valor crítico de QPS (peticiones por segundo)
Img-dnn	1300
Masstree	2500
Moses	150
Shore	70
Silo	2500
Specjbb	3000
Sphinx	1.5
Xapian	900

Tabla 6: Valores de QPS utilizados para el análisis de cada aplicación.

Empezamos en análisis estudiando el IPC de cada una de las aplicaciones. Un IPC bajo significa que la aplicación ejecuta pocas instrucciones por ciclo lo que habitualmente es debido a una mayor tasa de accesos a memoria principal ya que estos accesos suelen bloquear el ROB y detener la ejecución de la aplicación hasta que se resuelven. En cambio, un IPC alto suele significar que la aplicación realiza pocos accesos a memoria y el núcleo tiene una utilización más alta. La Figura 12 muestra el IPC para cada aplicación. Es interesante observar que las cuatro aplicaciones *SMT-friendly* son las que tienen un IPC más bajo mientras que las cuatro aplicaciones N*o SMT-friendly* tienen un IPC más alto.

Esta observación confirma que las aplicaciones con un IPC más bajo tienen una utilización menor de los recursos del núcleo. Esto permite que dos los dos hilos del servidor puedan ejecutarse en el mismo núcleo SMT con una interferencia baja y sin afectar negativamente a sus prestaciones. De esta forma, podemos utilizar menos núcleos para ejecutar el mismo número de hilos y, por tanto, los núcleos libres pueden emplearse para ejecutar otras aplicaciones. Cuando las aplicaciones tienen un IPC alto su utilización de los recursos es mayor y juntar los dos hilos del servidor en el mismo núcleo afecta a las prestaciones. Por tanto, cuando las aplicaciones tienen un IPC alto, como ya hemos visto, es mejor asignar cada hilo a un núcleo diferente para no reducir las prestaciones.

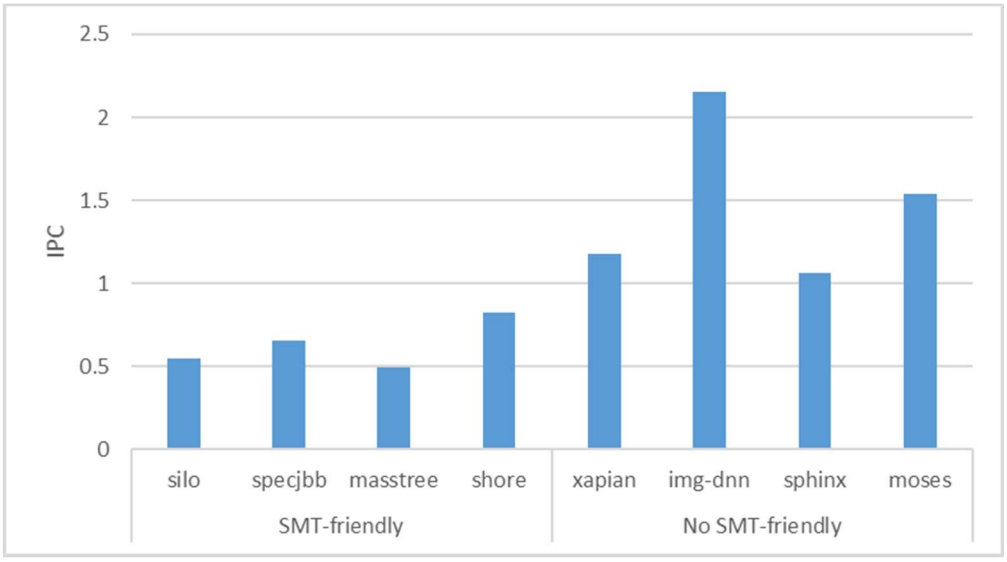

Figura 12: IPC alcanzado por las aplicaciones estudiadas utilizando un hilo.

La Figura 13 muestra el MPKI de L3 de las aplicaciones y la Figura 14 su tasa de acceso a memoria por microsegundo. Estos resultados confirman que el motivo por el que specjbb, masstree y shore alcanzan un IPC más bajo es debido a los accesos a memoria que realizan. Como ya hemos comentado, estas aplicaciones realizan una utilización más baja de los recursos del núcleo y por tanto son más afines a la ejecución SMT sin que sus prestaciones se vean degradadas.

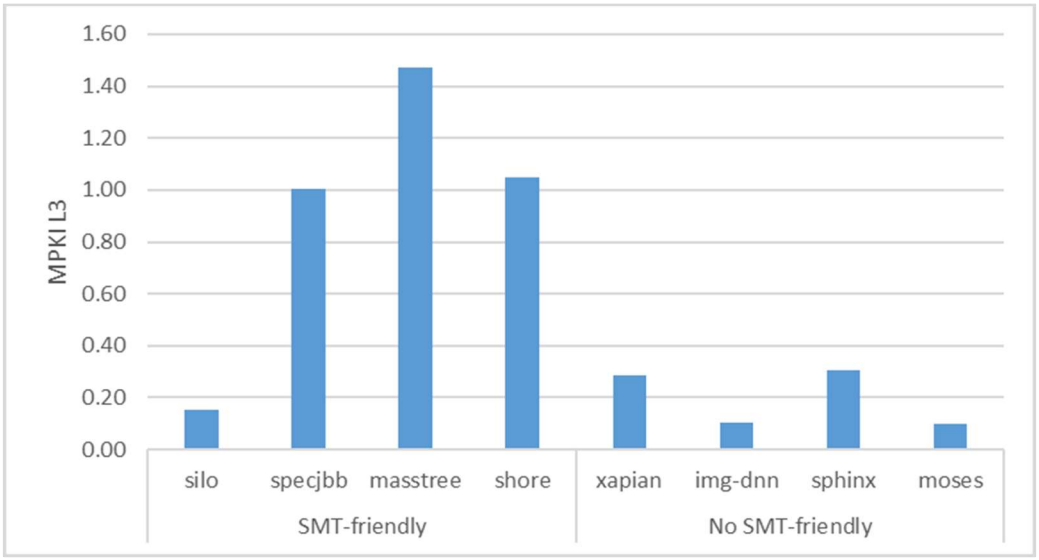

Figura 13: MPKI L3 para las aplicaciones estudiadas en la configuración de un hilo.

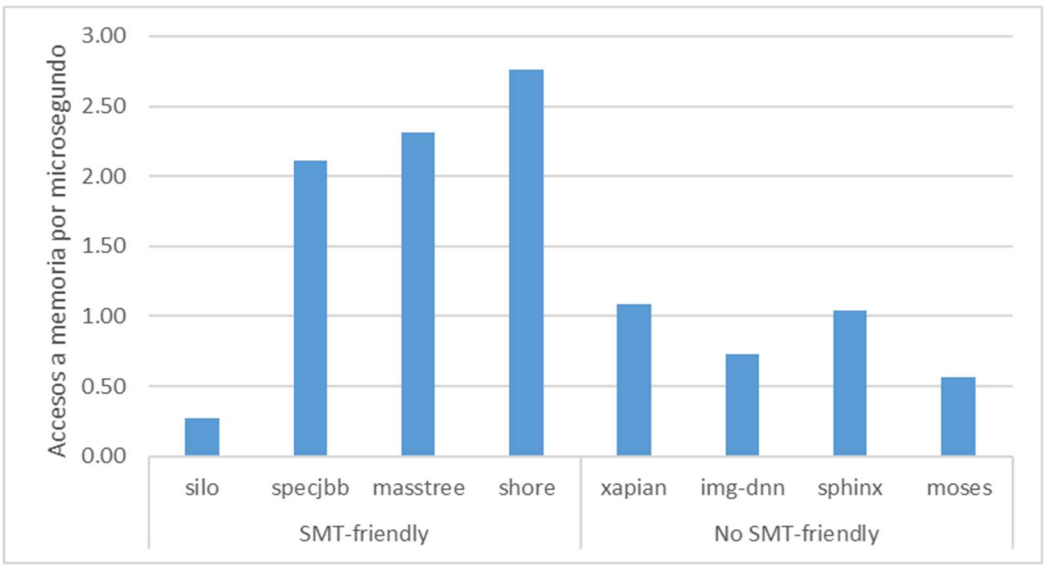

Figura 14: Accesos a memoria principal por microsegundo para las aplicaciones estudiadas en a la configuración de un hilo.

Otra métrica que podemos analizar para explicar el IPC bajo de las aplicaciones *SMT-friendly* son los ciclos de parada o stalls. Los contadores de prestaciones nos permiten monitorizar los ciclos de parada en la etapa dispatch del procesador. Es decir, nos permiten estudiar los ciclos en los que ninguna instrucción puede hacer dispatch. Evidentemente, no poder hacer dispatch significa retrasar la ejecución de todas las instrucciones que queden bloqueadas.

Aunque no es el único motivo que provoca los ciclos de parada, tener una tasa de aciertos baja en la LLC y realizar muchos accesos a memoria puede hacer que, por ejemplo, el ROB se bloquee y el procesador sufra ciclos de parada. La Figura 15 muestra el porcentaje de ciclos de parada para cada una de las aplicaciones estudiadas. Como

podíamos esperar, las aplicaciones *SMT-friendly* son las que tiene un mayor porcentaje de ciclos de parada.

Como ya hemos explicado, los ciclos de parada hacen que el procesador se bloquee. Por tanto, las aplicaciones con muchos ciclos de parada alcanzan buenas prestaciones cuando el servidor se ejecuta con dos hilos que comparten el núcleo SMT ya que un único hilo no puede aprovechar todos los recursos del núcleo. En cambio, cuando las aplicaciones tienen pocos ciclos de parada sí que están realizan una utilización alta de los recursos del núcleo. Por ello, estas aplicaciones, aunque mejoran sus prestaciones cuando se ejecutan con dos hilos en el mismo núcleo SMT respecto a utilizar un solo hilo, alcanzan mayores prestaciones cuando los hilos se ejecutan en núcleos distintos.

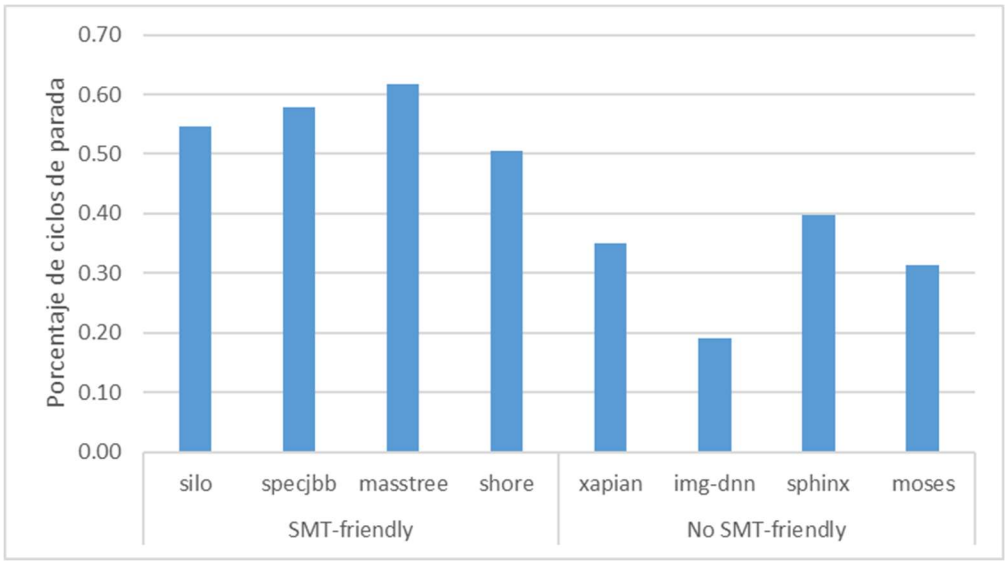

Figura 15: Ciclos de parada en la etapa de dipatch para las aplicaciones en la configuración de un hilo.

Finalmente, analizamos el porcentaje de ciclos de parada que son provocados por el reorder-buffer (ROB), por las estaciones de reserva (RS), por el *store buffer* (SB) y por otros recursos (por ejemplo, la *load queue*). El resto de ciclos de parada en la etapa de dispatch son provocados por otras causas, principalmente asociadas al front-end como fallos de la cache de instrucciones o de predicción de saltos. La Figura 16 muestra la clasificación de los ciclos de parada de las aplicaciones en función de la causa que los provoca.

La figura muestra que las causas que provocan los ciclos de parada en cada aplicación son diversas. Aunque antes vimos que las aplicaciones *SMT-friendly* sufrían más ciclos de parada que las aplicaciones *No SMT-friendly*, no vemos que ninguna causa predomine en todas las aplicaciones *SMT-friendly*. La observación más interesante es quizás la que explica el bajo IPC, bajo MPKI de L3 y baja tasa de accesos a memoria de silo: sus ciclos de parada son provocados por la alta utilización del store buffer. En el resto de aplicaciones los ciclos de parada se reparten de forma diversa, siendo las las estaciones de reserva, el ROB y las otras causas los motivos más habituales que

provocan ciclos de parada. También es llamativo el caso de sphinx, en el que el 80% de los ciclos de parada son provocados por las estaciones de reserva.

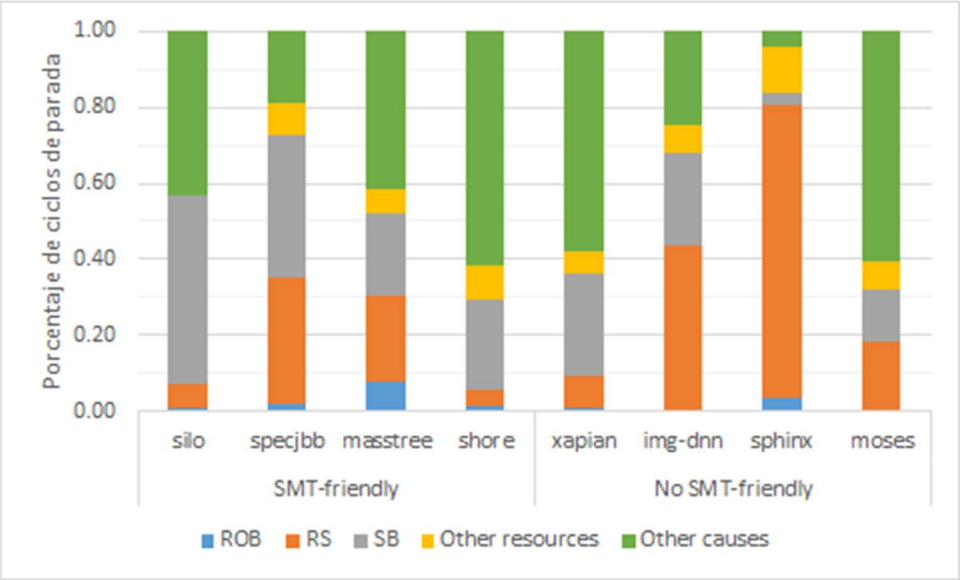

Figura 16: Clasificación de los ciclos de parada en función de la causa para las aplicaciones en la configuración de un hilo.

7. Conclusiones

La irrupción de la computación en la nube ha supuesto un cambio drástico sobre los sistemas de computación tradicional. Sus ventajas incitan a pensar que este tipo de computación cobrará todavía más importancia en los próximos años reemplazando a mucha de la computación tradicional. Con ella han cobrado especial relevancia nuevas aplicaciones con requisitos muy específicos. Hoy en día, las aplicaciones de latencia crítica son cada vez más comunes y la principal métrica que evalúa las prestaciones de estas cargas es la latencia de cola (habitualmente latencia de cola del percentil 95). Un empeoramiento en la latencia de cola significa una peor calidad de servicio, algo que los proveedores de servicios cloud intentan evitar a toda costa.

En este trabajo, hemos estudiado principalmente cómo el paradigma multi-hilo *simultáneos multithreading* (SMT) afecta el rendimiento de las aplicaciones de latencia crítica. Con este fin, hemos seleccionado la suite TailBench, que es una suite muy representativa y que incluye ocho aplicaciones de latencia crítica que cubren una amplia variedad de aplicaciones y requisitos muy diversos. Y hemos evaluado sus prestaciones en un sistema con un procesador Intel con tecnología de SMT (Hyper-threading).

La principal contribución del trabajo ha sido identificar dos tipos de comportamiento cuando se lanza el servidor con dos hilos y se ejecutan los dos hilos en el mismo núcleo SMT en comparación con ejecutarlos en dos núcleos distintos. En la mitad de las aplicaciones, ejecutar los dos hilos en el mismo núcleo SMT alcanza mejores prestaciones. Este sería el comportamiento esperado ya que se utilizan dos núcleos en lugar de uno. Sin embargo, en la otra mitad de aplicaciones, ejecutar los dos hilos en el mismo núcleo SMT alcanza las mismas prestaciones que ejecutarlos en dos núcleos distintos. Estos son debido a que algunas aplicaciones sufren muchos ciclos de parada y tienen una utilización baja de los recursos del núcleo. Las aplicaciones con este comportamiento permiten ejecutar dos hilos en el mismo núcleo SMT sin afectar negativamente a las prestaciones. En cambio, cuando las aplicaciones tienen una utilización más alta de los recursos del núcleo, se alcanzan mayores prestaciones cuando los hilos se ejecutan en núcleos distintos.

La investigación desarrollada en este TFM se ha centrado desde el punto de vista de los proveedores de la nube, que deben tener en cuenta las características de las aplicaciones y como les afecta la ejecución SMT para optimizar el rendimiento de los servidores que implementen procesadores con soporte SMT.

8. Bibliografía

[1] M. Peter, G.Tim, “The NIST Definition of Cloud Computing,” NIST Special Publication (SP) 800-145. National Institute of Standards, 2011.

[2] “Informe técnico sobre el desarrollo de la informática en la nube,” Academia de Tecnología de la Información y las Comunicaciones de China, 2020.

[3] Ren Rui, Ma Jiuyue, Sui Xiufeng,etc. “A Distributed Deadline Propagation Approach to Reduce Long-Tail in Datacenters,” Journal of computer research and development, vol. 54, no. 7, pp. 1617-1628, 2017.

[4] R. Nishtala, P. Carpenter, V. Petrucci and X. Martorell, "Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads," in Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, 2017, pp. 409-420.

[5] H. Kasture and D. Sanchez, "TailBench: a benchmark suite and evaluation methodology for latency-critical applications," in Proceedings of the International Symposium on Workload Characterization (IISWC), 2016, pp. 1-10.

[6] Manadhata, Pratyusa and V. Sekar. “Simultaneous Multithreading,” 2003.

[7] A. Agarwal et al., “The MIT Alewife machine: architecture and performance,” in Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA), 1995, pp. 2–13.

[8] W.-D. Weber and A. Gupta, “Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results,” in Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA), 1989, pp. 273–280.

[9] C. McNairy and R. Bhatia, “Montecito: a dual-core, dual-thread Itanium processor,” IEEE Micro, vol. 25, no. 2, pp. 10–20, 2005.

[10] Chaudhry, S. et al., “Simultaneous speculative threading: A novel pipeline architecture implemented in Sun’s Rock processor,” in Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA), 2009, pp. 484–495.

[11] A. Snavely and D. M. Tullsen, “Symbiotic jobscheduling for a simultaneous mutlithreading processor,” SIGPLAN Not., vol. 35, no. 11, pp. 234-244, 2000.

[12] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit, “Symbiotic job scheduling on the IBM POWER8,” in Proceedings of the International Symposium on High Performance Computer Architecture (HPCA), 2016, pp. 669–680.

[13] License, No and Intel Disclaims, “Intel® 64 and IA-32 Architectures Software Developer’s Manual,” 2006

[14] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler, “IBM POWER8 processor core microarchitecture,” IBM Journal of Research and Development, vol. 59, no. 1, pp. 2:1–2:21, 2015.

[15] S. K. Sadasivam, B. W. Thompto, R. Kalla and W. J. Starke, "IBM Power9 Processor Architecture," IEEE Micro, vol. 37, no. 2, pp. 40-51, 2017.

[16] Calzarossa, M. and G. Serazzi, “Workload characterization: a survey,” in Proceedings of the IEEE, vol. 81, no. 8, pp. 1136-1150, 1993.

[17] Cirne, W. and F. Berman, “A comprehensive model of the supercomputer workload,” 2001.

[18] Crovella, M, “Performance Evaluation with Heavy Tailed Distributions,” in Proceedings of the Computer Performance Evaluation. Modelling Techniques and Tools, 2000, pp. 1-9.

[19] Barford, P. and M. Crovella, “Generating representative Web workloads for network and server performance evaluation,” in Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems (SIGMETRICS '98/PERFORMANCE '98), 1998, pp. 151-160.

[20] Downey, A. and D. Feitelson, “The elusive goal of workload characterization,” SIGMETRICS Perform. Evaluation Rev, vol. 26, no. 4, pp. 14-29, 1999.

[21] Chen, Yanpei, A. Ganapathi, R. Griffith and R. Katz., “Towards Understanding Cloud Performance Tradeoffs Using Statistical Workload Analysis and Replay,” 2010.

[22] Mishra, A. K., J. L. Hellerstein, W. Cirne and C. Das, “Towards characterizing cloud backend workloads: insights from Google compute clusters,” SIGMETRICS Perform. Evaluation Rev, vol. 37, no. 4, pp. 34-41,

2010.

[23] Gmach, D., J. Rolia, L. Cherkasova and A. Kemper, “Workload Analysis and Demand Prediction of Enterprise Data Center Applications,” in Proceedings of the 10th International Symposium on Workload Characterization, 2007, pp. 171-180.

[24] Ganapathi, A., Yanpei Chen, A. Fox, R. Katz and D. Patterson, “Statistics-driven workload modeling for the Cloud,” in Proceedings of the 26th International Conference on Data Engineering Workshops (ICDEW), 2010, pp. 87-92.

[25] Govindan, S., Jeonghwan Choi, B. Urgaonkar, A. Sivasubramaniam and A. Baldini, “Statistical profiling-based techniques for effective power provisioning in data centers,” in Proceedings of the 4th ACM European Conference on Computer Systems, 2009, pp. 317-330.

[26] Bobroff, N., A. Kochut and K. Beaty, “Dynamic Placement of Virtual Machines for Managing SLA Violations,” in Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management, 2007, pp. 119-128.

[27] Verma, Akshat and Gargi Dasgupta, “Server Workload Analysis for Power Minimization using Consolidation,” in Proceedings of the USENIX Annual Technical Conference, 2009, pp. 28.

[28] Rolia, J., L. Cherkasova, M. Arlitt and A. Andrzejak, “A capacity management service for resource pools,” in Proceedings of the 5th International Workshop on Software and Performance, 2005, pp. 229-237.

[29] Chen, Gong, Wenbo He, J. Liu, S. Nath, Leonidas Rigas, Lin Xiao and F. Zhao. “Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services,” in Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, 2008, pp. 337-350.

[30] Park, J., Seongbeom Park and W. Baek, “CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers,” in Proceedings of the Fourteenth EuroSys Conference, 2019, pp. 1-16.

[31] Park, J., S. Park, M. Han, Jihoon Hyun and W. Baek, “Hypart: a hybrid technique for practical memory bandwidth partitioning on commodity servers,” in Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, 2018, pp. 1-14.

[32] Lo, David, Liqun Cheng, R. Govindaraju, L. Barroso and C. Kozyrakis. “Towards energy proportionality for large-scale latency-critical workloads,” in

Proceeding of the 41st Annual International Symposium on Computer Architecuture, 2014, pp. 301-312.

[33] Lo, David, Liqun Cheng, R. Govindaraju, P. Ranganathan and C. Kozyrakis, “Heracles: Improving resource efficiency at scale,” in Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015, pp. 450-462.

[34] Schwarzkopf, M., A. Konwinski, M. Abd-El-Malek and J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” in Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 351-364.

[35] Verma, A., Luis Pedrosa, Madhukar Korupolu, D. Oppenheimer, E. Tune and J. Wilkes, “Large-scale cluster management at Google with Borg,” in Proceedings of the Tenth European Conference on Computer Systems, 2015, pp. 1-17.

[36] Delimitrou, Christina and C. Kozyrakis, “HCloud: Resource-Efficient Provisioning in Shared Cloud Systems,” in Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, 2016, pp. 473-488.

[37] Delimitrou, Christina, D. Sánchez and C. Kozyrakis, “Tarcil: reconciling scheduling speed and quality in large shared clusters,” in Proceedings of the Sixth ACM Symposium on Cloud Computing, 2015, pp. 473-488.

[38] Mars, Jason and L. Tang, “Whare-map: heterogeneity in "homogeneous" warehouse-scale computers,” in Proceedings of the 40th Annual International Symposium on Computer Architecture, 2013, pp. 619-630.

[39] Delimitrou, Christina and C. Kozyrakis, “Bolt: I Know What You Did Last Summer... In The Cloud,” in Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, 2017, pp. 599-613.

[40] Kasture, Harshad, D. Bartolini, Nathan Beckmann and D. Sánchez, “Rubik: Fast analytical power management for latency-critical systems,” in Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48), 2015, pp. 598-610.

[41] Kasture, Harshad and D. Sánchez, “Ubik: efficient cache sharing with strict qos for latency-critical workloads,” in Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, 2014, pp. 729-742.

[42] Sánchez, D. and C. Kozyrakis, “Vantage: Scalable and efficient fine-grain cache partitioning,” in Proceedings of the 38th annual international

symposium on Computer architecture, 2011, pp. 57-68.

[43] Zhang, Xiao, E. Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale and J. Wilkes, “CPI2: CPU performance isolation for shared compute clusters,” in Proceedings of the 8th ACM European Conference on Computer Systems, 2013, pp. 379-391.

[44] S. Chen, S. GalOn, C. Delimitrou, S. Manne and J. F. Martínez, "Workload characterization of interactive cloud services on big and small server platforms," in Proceedings of the International Symposium on Workload Characterization (IISWC), 2017, pp. 125-134.

[45] A. Khan, X. Yan, S. Tao and N. Anerousis, "Workload characterization and prediction in the cloud: A multiple time series approach," in Proceedings of the Network Operations and Management Symposium, 2012, pp. 1287-1294.

[46] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA), 2015, pp. 450-462.

[47] Shuang Chen, Christina Delimitrou, and José F. Martínez, “PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services,” in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 107–120.

[48] Linux Containers. Accessed: Feb. 10, 2021. [Online]. Available: https://linuxcontainers.org/

[49] Ghods, A. R, “A Study of Linux Perf and Slab Allocation Sub-Systems,” UWSpace, 2016. [Online]. Available: http://hdl.handle.net/10012/10184

[50] Xmodulo – Linux FAQ. Accessed: Jan. 20, 2021. [Online]. Available: https://www.xmodulo.com/run-program-process-specific-cpu-cores-linux.html

[51] Die.net. Accessed: Dec. 14, 2020. [Online]. Available: https://linux.die.net/man/1/taskset

[52] ArnoldLu. “Introduction and use of system-level performance analysis tool perf." https://www.cnblogs.com/arnoldlu/p/6241297.html (accessed Jun. 14, 2020).

[53] Shaoyuan Li. “Linux performance analysis tool: Perf.” http://wiki.csie.ncku.edu.tw/embedded/perf-tutorial (accessed Jun. 14, 2020).

[54] 726561. “Linux performance counter is a new kernel-based subsystem.” https://www.cnblogs.com/phploser/p/12397851.html (accessed Jun. 20, 2020)