Performance y escalabilidad del kernel Linux aplicado a redes de alta velocidad

Abril de 2007

Resumen

Asumiendo un futuro cercano donde la computación centrada en las redes y las arquitecturas paralelas serán características fundamentales, se hace evidente la necesidad de replantear el diseño del software de base teniendo en cuenta los nuevos paradigmas de procesamiento y características del hardware. Por un lado, se espera un gran desarrollo tecnológico basado en las distintas formas de almacenamiento y procesamiento distribuido, soportado por el incesante avance en los estándares y tecnologías de redes de datos. Por otro lado, las arquitecturas con múltiples núcleos de procesamiento se están posicionando como el estándar actual para la mayoría de las configuraciones de hardware.

En este escenario, se están planteando nuevas características (y adaptando antiguas ideas) que los sistemas operativos deberán incorporar en el corto y mediano plazo. Entre otras, se puede mencionar el diseño de kernels asimétricos y activos para explotar al máximo el paralelismo en las nuevas arquitecturas. En este informe se describe una modificación al diseño del subsistema de red del kernel Linux 2.6 que permite evaluar el efecto de hacer un uso asimétrico de los procesadores en una máquina con arquitectura SMP asignando un procesador de manera exclusiva al procesamiento de red. De esta forma se intenta reducir el costo conocido como intrusión del sistema operativo.

1. Introducción: La intrusión del sistema operativo

La intrusión es un nombre nuevo para un concepto que existe desde que las computadoras basan su funcionamiento en un proceso monitor o sistema operativo, y representa el costo (medido en procesamiento y almacenamiento) introducido por el kernel al realizar sus funciones principales de virtualización y protección de los recursos de la máquina. Las interrupciones por hardware y las llamadas al sistema implementadas como *traps* son un ejemplo de intrusión por mecanismo (intrusión relacionada con la implementación de las funciones del sistema operativo).

En general, el nivel de intrusión tiene una relación directa con la manera en que las políticas y mecanismos del sistema operativo se adaptan al hardware subyacente y a la forma en que las aplicaciones lo utilizan. Mientras que estos dos factores han evolucionado notablemente en las últimas décadas, los sistemas operativos más utilizados en la actualidad se basan en conceptos introducidos hace aproximadamente tres décadas (un intervalo de tiempo enorme en lo que se refiere a tecnología informática). Los mecanismos del kernel que deberían ser invocados excepcionalmente se vuelven demasiados costosos cuando se usan con frecuencias mucho más altas que aquellas para las cuales fueron pensados. Por ejemplo, los altos costos pueden surgir al utilizar sistemas operativos diseñados para tareas intensivas en computación (esto es, no

optimizados para la entrada/salida) en máquinas cuya principal función es el movimiento de datos, como es el caso de un *Storage Element* en una infraestructura Grid.

Por ejemplo, en los modelos tradicionales de interfaces de red, el procesador es interrumpido por cada paquete recibido del enlace. Sin embargo, las interfaces de alta velocidad pueden recibir miles de paquetes por segundo[1], generando miles de interrupciones (y, en consecuencia, miles de intercambios de tareas) por segundo. Además de utilizar ciclos de procesador, las interrupciones asincrónicas y cambios de contextos frecuentes, también causan efectos indirectos como la polución de la memoria cache y del *Translation Lookaside Buffer* (TLB), lo que incrementa aun más el costo debido al procesamiento de red.

Los sistemas Chip Multiprocessing (CMP) y Symmetric Multiprocessing (SMP) tienen más probabilidades de sufrir intrusión por mecanismo, ya que la mayoría de los sistemas operativos de propósito general utilizados en estas máquinas, fueron adaptados de una base monoprocesador. Por lo tanto, además de coordinar el acceso de múltiples aplicaciones independientes a un único recurso físico, el sistema operativo debe propagar esta protección a lo largo de múltiples CPUs.

2. La intrusión generada por el procesamiento TCP/IP en Linux sobre SMP

2.1. Mecanismos de Linux involucrados en el procesamiento de red

A continuación vemos como se utilizan algunos conceptos de threading en el procesamiento TCP/IP de Linux. Existe un tipo de thread especial que procesa los protocolos registrados en el kernel y se ejecuta concurrentemente con los manejadores de interrupciones de los drivers de las NICs (Network Interface Controllers). También se presentan algunas estructuras de datos que influyen significativamente en la forma de procesar el tráfico de red en Linux.

2.1.1. Interrupciones y funciones diferidas

Dos de los mecanismos esenciales en la implementación del subsistema de red en el kernel Linux (y en la mayoría de los sistemas operativos modernos) son el soporte de las interrupciones por hardware y la implementación de las funciones diferidas en el tiempo, conocidas generalmente como softirqs o bottom halves (no se deben confundir con las interrupciones por software, también conocidas como traps).

Cuando una señal de interrupción llega a la CPU, el kernel debe realizar ciertas tareas que generalmente son implementadas desde el manejador de esa interrupción. No todas las acciones a ser realizadas tienen la misma urgencia. De hecho, el manejador de interrupciones no es un entorno en el que puedan realizarse cualquier tipo de acción (por ejemplo, no se puede dormir a un proceso). Las operaciones largas y no críticas deberían ser diferidas ya que mientras el manejador se está ejecutando, el kernel está en lo que se denomina como contexto de interrupción (se le llama top half al manejador que se ejecuta en el contexto de la interrupción, de ahi el nombre de bottom half para la otra parte del manejador) y la CPU que lo ejecuta tiene deshabilitadas todas las interrupciones. Es decir, los manejadores de interrupciones son no interrumpibles y no reentrantes (al menos hasta que el propio manejador activa nuevamente las interrupciones). Esta decisión de diseño ayuda a reducir la probabilidad de condiciones de carrera, sin embargo tiene efectos serios en la performance si no se tiene cierto cuidado [2].

[1] A una velocidad de 10Gbps, con tramas de 1500 bytes (12000 bits), se reciben 833.333 tramas por segundo. Esto significa casi 1 millon de interrupciones de recepción de red por segundo, si no se utiliza algún mecanismo de reducción de interrupciones.

[2] Algunas implementaciones primitivas de TCP/IP o diseñadas para entornos embebidos completan el procesamiento

La cantidad de procesamiento requerido por una interrupción depende del tipo de evento que se esta señalizando. En particular, los dispositivos de red requieren un trabajo relativamente complejo: en el peor de los casos[3] se requiere alocar un buffer, copiar los datos recibidos en el mismo, inicializar algunos parámetros del buffer para indicar el protocolo de capa dos a las capas superiores, hacer el procesamiento de capas superiores, copiar los datos al buffer del usuario, etc. Por lo tanto, y más aun con las interfaces de red de alta velocidad actuales, es importante consumir la menor cantidad de tiempo posible en los manejadores de interrupciones, o el sistema sufrirá una perdida de capacidad de respuesta.

Además, TCP/IP, como otros protocolos de red, es de naturaleza asincrónica. Básicamente, se implementa como una gran máquina de estados basada en un conjunto de timers. La cantidad de tiempo que se necesita para procesar un paquete y enviarlo a la capa de aplicación depende del protocolo de transporte y de muchos otros factores. La implementación TCP/IP de Linux provee buffering a varios niveles o capas, de forma tal que un componente no retrasará a la pila entera. Por estos motivos, es preferente permitir que la pila TCP/IP procese los paquetes entrantes independientemente del driver.

Aquí es donde el concepto de softirq entra en juego. Incluso si las acciones disparadas por una interrupción necesitan mucho tiempo de CPU, la mayoría de estas acciones usualmente puede esperar. Se define una softirq como un requerimiento asincrónico para ejecutar una función en particular[4]. Este modelo permite que el kernel mantenga las interrupciones deshabilitadas por mucho menos tiempo, y funciona siguiendo estos pasos:

- El dispositivo señaliza a la CPU para notificar de un evento.
- La CPU ejecuta el top half asociado, el cual típicamente realiza las siguientes tareas:
  - Guarda en la RAM toda la información que el kernel necesitará más tarde para procesar el evento que interrumpió a la CPU,
  - planifica la softirq que se encargará de procesar finalmente la interrupción con los datos almacenados por el manejador,
  - y por último rehabilita las interrupciones para la CPU local.
- En algún momento en el futuro cercano, se chequea por softirqs planificadas y se invoca a los manejadores correspondientes en caso de ser necesario.

A lo largo del tiempo, los desarrolladores de Linux han intentado diferentes tipos de mecanismos de bottom halves, las cuales obedecen a diferentes reglas (principalmente reglas de concurrencia y de contexto de ejecución). El subsistema de red ha jugado un rol central en el desarrollo de nuevas implementaciones por sus requerimientos de baja latencia, es decir una cantidad mínima de tiempo entre la recepción de una trama y su entrega a las capas superiores. La baja latencia es más importante para los drivers de dispositivos de red que para otros tipos de drivers ya que si las tramas comienzan a acumularse en la cola de recepción sin que se ejecuten las softirqs de red, se empezarán a descartar las tramas entrantes.

Gran parte del procesamiento de red en el kernel Linux se hace en el contexto de dos softirqs que se registran en la inicialización del subsistema de red y serán planificadas cada vez que se reciba o se transmita una trama. Estas softirq son conocidas como: NET_TX_SOFTIRQ y NET_RX_SOFTIRQ y sus manejadores son las

---

de un paquete entrante en el contexto del manejador de interrupción. Sin embargo los sistemas operativos modernos suelen utilizar estas capacidades de threading.

[3] Puede haber optimizaciones para cada uno de estos pasos.

[4] Actualmente se utiliza la denominación de bottom half para hacer referencia al concepto de función diferida, aunque se implemente con otros mecanismos, como las softirqs y los tasklets.

funciones net_tx_action() (tareas relacionadas con paquetes salientes) y net_rx_action() (procesamiento de paquetes entrantes). Cuando el kernel debe manejar gran cantidad de tráfico de red, ocupa gran parte de la CPU con estas funciones, por eso es importante conocer los efectos secundarios que producen estos mecanismos.

Es conveniente mencionar algunas reglas y observaciones importantes con respecto a las softirq. Primero, se planifican y ejecutan en la misma CPU que recibió la interrupción del dispositivo y, aunque se serializan por CPU, en una máquina SMP puede haber varias activaciones del mismo manejador de softirq al mismo tiempo en distintas CPU, es decir, deben ser funciones reentrantes. Por lo tanto, las softirq debe proteger explícitamente (generalmente lo hacen con spinlocks) las estructuras de datos a las que acceden.

Como se explica más adelante, el kernel 2.6 tiene una estructura por CPU, llamada softnet_data, que contiene información de red que debe accederse para cada trama entrante o saliente (colas, información de congestión, etc), atendida por esa CPU. De esta forma y como las softirq se serializan por CPU, no necesitan usar mecanismos de protección para acceder a esta información. Sin embargo, se deben proteger los accesos a muchas otras estructuras que son accedidas por todas las CPUs, como por ejemplo las estructuras relacionadas con el driver de las NICs y las estructuras que representan el estado de una conexión TCP, etc.

Por otro lado, una vez planificadas, los manejadores de softirqs se ejecutarán cuando el kernel verifique la existencia de softirqs pendientes. Esto se hace en varios lugares en el código del kernel, por ejemplo, cuando se retorna de una interrupción por hardware, cuando se reactivan las funciones diferibles (suelen desactivarse temporalmente por cuestiones de sincronización), cuando se despiertan los threads ksoftirqd (explicado más adelante), etc. Por lo tanto, en situaciones de alto tráfico de red, estas softirq serán intercaladas continuamente con los procesos de usuario, incrementando la polución de cache y TLB, cantidad de cambios de contextos, etc.

Por último, generalmente se discute la escalabilidad de este mecanismo. Al ser serializadas por CPU, las softirq NET_RX_SOFTIRQ y NET_TX_SOFTIRQ representan los únicos 2 contextos en los que se estarán procesando protocolos de red por cada CPU. Es decir, cada procesador dispone de un thread de transmisión y uno de recepción que se ejecuta intermitentemente, junto con el resto de las funciones del sistema y los procesos de usuario. Sin embargo, se mencionó anteriormente que una CPU puede estar mucho tiempo paralizada por la latencia en los accesos a memoria principal y estos ciclos ociosos no son aprovechados para realizar trabajo útil sobre otros paquetes. Básicamente, el procesamiento de paquetes, en una dirección (entrada o salida) y por CPU, es secuencial. Intel ya está trabajando en una pila que mejora este comportamiento agregando concurrencia de granularidad mucho más fina [18].

2.1.2. La estructura softnet_data

Uno de las mejoras más importantes para la escalabilidad de red en Linux, fue la de distribuir las colas de ingreso y egreso de paquetes de red entre las CPUs. Como cada CPU tiene su propia estructura de datos para manejar el tráfico entrante y saliente, no hay necesidad de utilizar locking entre las diferentes CPUs. La estructura de datos, softnet_data, se define de la siguiente forma:

```
struct softnet_data
{
    int             throttle;
    int             cng_level;
    int             avg_blog;
    struct sk_buff_head    input_pkt_queue;
    struct list_head       poll_list;
    struct net_device      *output_queue;
    struct sk_buff         *completion_queue;
    struct net_device      backlog_dev;
}
```

Incluye campos utilizados para la recepción y campos usados en la transmisión, por lo que tanto la softirq NET_RX_SOFTIRQ como

NET_TX_SOFTIRQ acceden a ella para obtener las tramas recibidas por la NIC y colocadas en esta estructura por el manejador de interrupciones. Las tramas de entrada son encoladas en input_pkt_queue (cuando no se usa NAPI, explicado más abajo) y las tramas de salida se colocan en colas especializadas que son administradas por el software de QoS (Traffic Control). La softirq de transmisión se usa para las retransmisiones y para limpiar los buffers de transmisión después de efectuado el envío, evitando el hacer más lenta la transmisión por esta operación.

En la versión 2.5, fue introducida una Nueva API para el manejo de tramas entrantes en el kernel Linux, conocida como NAPI. Como pocos drivers han sido actualizados para usar esta API, hay dos formas en que un driver en Linux puede notificar al kernel sobre una nueva trama:

Con el método tradicional, el driver maneja la interrupción de la NIC y encola la nueva trama en el miembro input_pkt_queue de la estructura softnet_data de la CPU que recibió la interrupción.

Usando NAPI, cuando una NIC recibe una trama el driver desactiva las interrupciones y la coloca en la lista apuntada por poll_list de la estructura softnet_data correspondiente. Luego será verificada en las subsiguientes invocaciones a net_rx_action hasta que no haya más tramas que recibir. Finalmente se reactivan las interrupciones. Éste método híbrido sólo funciona en la recepción y muy pocos drivers implementan el soporte necesario para usarlo.

2.1.3. Los socket buffers

Los socket buffers de Linux (sk_buff) son estructuras utilizadas por todas las capas del subsistema de red para referenciar paquetes y consiste en los datos de control y una referencia a la memoria donde se ubican los datos del paquete. Si son paquetes salientes, se alocan en la capa de sockets. Si son paquetes entrantes se alocan en el driver del dispositivo de red.

El pasaje de un paquete de una capa de la pila de protocolos a la próxima, se suele realizar a través de colas que se implementan utilizando listas doblemente enlazadas de socket buffers.

2.2. Identificando componentes del procesamiento TCP/IP en Linux 2.6

Para estudiar los efectos de la intrusión del sistema operativo en un caso concreto, se analizó el comportamiento de la pila TCP/IP implementada en el kernel Linux 2.6 cuando se ejecuta sobre una arquitectura SMP. El procesamiento de red multigigabit puede llegar a consumir hasta un 80 % de la capacidad de cómputo en máquinas modernas [8, 16] y gran parte de este porcentaje se debe a la intrusión.

Como se muestra en la figura 1 (adaptada de [15]), se pueden identificar cinco componentes distintos en el procesamiento tradicional de TCP/IP. Es interesante tener presente esta división funcional por varios motivos. Por un lado, permite identificar con precisión la fuente del overhead generado por las operaciones de red. Pero aún más importante, es tener la posibilidad de plantear diferentes alternativas de distribución del procesamiento de los componentes entre CPUs. En la sección donde se explica el prototipo implementado, se verá que algunos componentes del procesamiento TCP/IP se harán de manera exclusiva en un procesador dedicado a tal fin.

- Procesamiento de interrupciones: El componente de procesamiento de la interrupción incluye el tiempo que toma manejar las interrupciones de la NIC y setear las transferencias DMA, tanto en la transmisión como en la recepción. En el camino de recepción, una interrupción es elevada cuando una trama arriba a la interface y en la transmisión el hardware indica con una interrupción que el envío se completó, entonces el driver libera o recicla el sk_buff utilizado en la operación.

- Recepción inferior: El próximo componente es el procesamiento de recepción que comienza al obtener los paquetes de la estrutc-

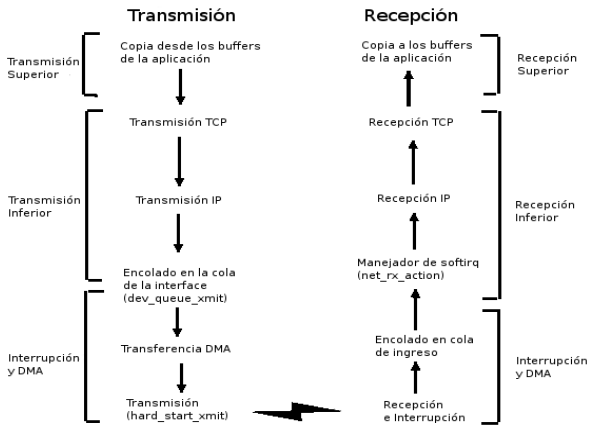

Figura 1: Componentes del procesamiento TCP/IP

tura softnet_data y continúa hasta encolarlos en la cola del socket. No incluye el tiempo que toma copiar los datos desde el socket buffer al buffer de la aplicación. Una vez que el procesamiento de la interrupción se completa, el manejador de la softirq de recepción, net_rx_action(), desencola paquetes de la cola de ingreso y los pasa al manejador de paquetes apropiado (packet handler). En el caso de TCP/IP, el manejador de paquetes es es la función ip_rcv(). Se completa el procesamiento de recepción IP y se invoca al protocolo de transporte en tcp_rcv(). Luego se encola el sk_buff en la cola del socket apropiado.

- Recepción superior: Aqui se explica la porción de la recepción que copia los datos al buffer de la aplicación. Luego que los datos alcanzan la cola del socket, y cuando la aplicación invoca a la system call recv() o read(), los datos son copiados en los buffers de la aplicación desde el buffer del kernel. Si la aplicación invoca a recv() antes que haya datos disponibles en el socket, la aplicación es bloqueada y posteriormente, cuando los datos arriban, se despierta al proceso.

- Transmisión superior: Este componente se refiere a la porción del procesamiento de envío que copia los datos del buffer de la aplicación al buffer en el kernel. Cuando la aplicación invoca a send(), se traduce en una llamada a la función tcp_sendmsg(), la cual aloca un sk_buff, se construyen los headers TCP e IP y luego copia los datos desde el buffer de la aplicación. Si los datos son más grandes que el MSS (Maximum Segment Size), los datos son copiados a múltiples segmentos TCP. Alternativamente, varios buffers de datos pequeños pueden ser empaquetados en un único segmento TCP.

- Transmisión inferior: Este componente se refiere al procesamiento de transmisión realizado luego de copiar los datos al buffer del kernel. Esto incluye calcular el checksum (aunque existe la posibilidad de combinar el cálculo del checksum con la copia desde el espacio de usuario) y agregar información específica de TCP a los headers. Dependiendo del estado de la conexión TCP, y los argumentos a la system call send(), TCP hace una copia de todo, alguna parte, o nada de los datos para posibilitar futuras restransmisiones. Luego se invoca la función de envío IP, ip_queue_xmit(), la cual agrega valores específicos del protocolo (incluyendo el checksum IP). Finalmente el paquete está listo para ser transmitido y se agrega a la cola de salida de la interface.

2.3. La escalabilidad de la implementación TCP/IP de Linux en arquitecturas SMP

Debido a la forma en que se distribuyen las interrupciones por hardware y la manera de planificar las softirqs de red [7, 12], se llega a la conclusión de que este kernel sigue el paradigma de paralelismo por mensaje [17]. Pero también se observa que este modelo no presenta un buen comportamiento con respecto a la escalabilidad.

Las pilas de protocolos de red, en particular las

implementaciones de TCP/IP, son conocidas por su incapacidad de escalar correctamente en los sistemas operativos monolíticos de propósito general sobre arquitecturas SMP. Trabajos previos [5, 10, 6, 4, 8] han explicado algunos de los problemas relacionados con el uso de múltiples CPUs para el procesamiento de red. El problema de escalabilidad no sólo se presenta al agregar CPUs, sino que también se encontraron problemas de gran overhead al agregar varias NICs de Gigabit a una máquina.

Para confirmar estas afirmaciones se efectuaron algunas pruebas sobre una máquina SMP con dos procesadores AMD Opteron y dos placas Gigabit Ethernet. La descripción de los experimentos, el conjunto de herramientas utilizadas y el método de instrumentación del kernel son detallados en la próxima sección, sin embargo resulta útil anticipar en esta sección algunos resultados que evidencian un problema de escalabilidad. El primer subconjunto de pruebas se realizó sobre el kernel Linux 2.6.17.8 sin modificaciones y se estresaron ambos enlaces para evaluar el costo en procesamiento e identificar como se distribuye ese costo. Ésto se intentó utilizando primero una única CPU y luego ambas CPUs. Los primeros resultados fueron notables: mientras el rendimiento de red se mantuvo prácticamente constante, el consumo total de CPU se incrementó del 50 % al 85 %.

En su última publicación[8], Van Jacobson afirma que “este comportamiento es simplemente la Ley de Amdahl en acción”. En este caso, cuando se agregan CPUs adicionales al procesamiento de red, el beneficio parece crecer linealmente pero el costo de contención, serialización, scheduling, etc., crece geométricamente.

Uno de los mayores impedimentos para la escalabilidad en las arquitecturas actuales es la performance de memoria. Por lo tanto el comportamiento de la cache de datos será frecuentemente el factor dominante en la performance del código del kernel. El comportamiento de la cache de instrucciones en un protocolo de red como TCP/IP también tiene un impacto significativo en la performance [14], este es el motivo por el cual, en ocasiones, simplemente reducir el tamaño del código ejecutable incrementa la performance, ya que incrementa la localidad de las referencias con respecto a la cache de instrucciones. Sin embargo, el código del subsistema de red realiza un conjunto de acciones que reducen la performance de la cache:

- El pasaje de los paquetes a través de múltiples contextos y capas del kernel: Cuando un paquete arriba a la máquina, el manejador top half de interrupción de la interface de red comienza la tarea de recibirlo. El procesamiento central de red sucede en la softirq net_rx_action(), es decir en otra función. La copia de los datos a la aplicación sucede en el contexto de una system call. Finalmente la aplicación realizará computación útil con los datos. Los cambios de contexto son costosos, y si alguno de estos cambios causa que el trabajo sea movido de una CPU a otra, se genera un gran costo de cache. Se ha realizado mucho trabajo para mejorar la localidad de CPU en el subsistema de red, pero queda mucho por hacer.

- El locking es muy costoso, disminuyendo la performance de dos formas: El lock tiene múltiples escritores por lo que cada uno tiene que efectuar una operación de cache del tipo RFO (Read-For-Ownership), que es extremadamente costosa. Además un lock requiere una actualización atómica lo que se implementa bloqueando la cache. Los costos del locking han llevado al desarrollo de técnicas libres de lock en el kernel 2.6, como seqlocks y read-copy-update[9], pero en el código de red todavía se utilizan locks como mecanismo principal.

- El código de networking hace uso intensivo de colas implementadas con listas circulares doblemente enlazadas. Estas estructuras generan un comportamiento pobre de cache ya que no gozan de localidad de las referencias y

requieren que cada usuario haga cambios en múltiples lugares. Ya que muchos componentes del subsistema de red tienen una relacion productor consumidor, una cola FIFO libre de locks puede realizar esta función con mayor eficiencia.

Jacobson piensa que la clave para una mayor escalabilidad de red es liberarse del locking y de los datos compartidos tanto como se pueda. Sin embargo, siempre que se continue distribuyendo el procesamiento de cada paquete de manera descontrolada entre todos los procesadores, el código de red no mostrará un comportamiento de cache adecuado. Además, aunque no son operaciones de costo intensivo, los costos incurridos por interrupciones y scheduling tienen un impacto indirecto en la efectividad de la cache. El scheduler en sistemas operativos típicos, siempre trata de balancear la carga, moviendo los procesos de procesadores con mayor carga a aquellos con menor carga. Cada vez que ocurre un scheduling o una interrupción que causa una migracion de proceso, el proceso migrado debe alimentar varios niveles de caches de datos en el procesador al cual fue migrado.

2.3.1. Afinidad de procesos e interrupciones a CPUs

Se han publicado artículos[11, 4] planteando la posibilidad de mejorar la performance de red a través del uso de afinidad de procesos/threads e interrupciones a CPUs, es decir la asociación controlada de determinados procesos e interrupciones de hardware a CPUs específicas.

La afinidad de procesos evita las migraciones innecesarias de procesos a otros procesadores. La afinidad de interrupciones se asegura que tanto el bottom half como el top half del manejador de interrupciones se ejecuten en el mismo procesador para todos los paquetes recibidos por una NIC. Con la afinidad total, las interrupciones son servidas en el mismo procesador que finalmente ejecutará las capas superiores de las pilas así como la aplicación del usuario. En otras palabras, hay un camino directo de ejecución en el procesador. Ya que la ejecución del código y el acceso a los datos son confinados al mismo procesador, se logra gran localidad de referencias.

Mientras la afinidad estática puede no funcionar para aplicaciones cambiantes dinámicamente y no uniformes, los servidores dedicados, por ejemplo un servidor web corriendo un número conocido de threads workers y NICs, pueden tener workloads que sean apropiados para este tipo de mecanismos. Dos web servers comunmente usados (TUX, de Red Hat, y IIS de Microsoft) ya permiten afinidad de procesos y threads.

Este concepto es tan importante que las NICs de la próxima generación tendrán la habilidad de observar dentro de los paquetes para extraer información de flujo y dirigir las interrupciones, dinámicamente, a los procesadores con la cache más acertada. De hecho, se cree que la afinidad tendrá un rol central en los sistemas operativos del futuro, para máquinas SMP/CMP, extendiendo estas ideas más alla del procesamiento de red.

3. Un prototipo básico y pruebas de performance

3.1. El prototipo

Generalmente, los sistemas operativos han utilizado a las máquinas SMP de manera simétrica, intentando balancear la carga entre todos los elementos de procesamiento disponibles. Sin embargo, la idea de un kernel activo [13] propone dedicar uno o varios CPUs (o núcleos) a realizar exclusivamente actividades del kernel, de manera tal de no depender de mecanismos costosos para la invocación de sus funciones. Esta sección presenta una modificación al subsistema de red del kernel Linux 2.6 basada en esta idea. Además se exponen algunas mediciones realizadas comparando una configuración del kernel estándar y un kernel modificado.

Con esta modificación se intenta desacoplar el

procesamiento de las aplicaciones de usuario del procesamiento de red. El procesamiento TCP/IP es aislado de las CPUs que ejecutan el sistema operativo y los procesos de usuario (CPUs-HOST), y se delega a un procesador o subconjunto de procesadores dedicados a las comunicaciones (CPUs-NET). Las CPUs-HOST deben comunicarse con las CPUs-NET usando un medio de comunicación de bajo costo y no intrusivo, como la memoria compartida.

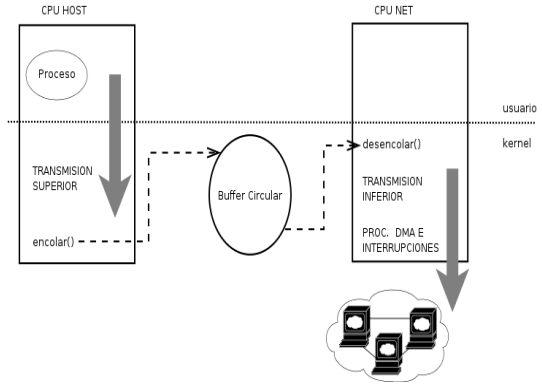

Figura 2: Proceso de transmisión del prototipo

Una CPU-NET no necesita ser multiplexada entre otras aplicaciones y/o funciones del kernel, con lo cual puede ser optimizada para lograr la mayor eficiencia posible en su función. Por ejemplo, es posible evitar el costo de las interrupciones de hardware si las CPUs-NET consultan, con una frecuencia suficiente, el estado de las NICs para saber si necesitan ser atendidas. También se reduce la contención por datos compartidos y el costo de sincronización dado que el procesamiento de red ya no se realiza de manera asincrónica y descontrolada entre todas las CPUs de la máquina, sino que se ejecuta sincrónicamente en un subconjunto reducido de CPUs (posiblemente sólo una CPU, dependiendo del tráfico a procesar).

De esta forma se puede incrementar la eficiencia en el procesamiento de los protocolos de red al reducir la intrusión por mecanismo del sistema operativo. Las aplicaciones ya no serán interrumpidas varias miles de veces por segundo y el scheduler no sufrirá las interferencias causadas por los altos requerimientos de procesamiento que implica el tráfico de red de alta performance. Es decir, sabiendo que las interrupciones de las NICs y las softirq correspondientes pueden consumir más de una CPU en una máquina SMP, directamente se aísla una CPU para ejecutar este código de manera sincrónica y controlada. Con esto se intenta reducir el tiempo consumido por las funciones del kernel que no realizan trabajo útil para el usuario, como son las funciones del subsistema de interrupciones, las que planifican y disparan las softirq, y las que realizan los cambios de contexto, entre otras.

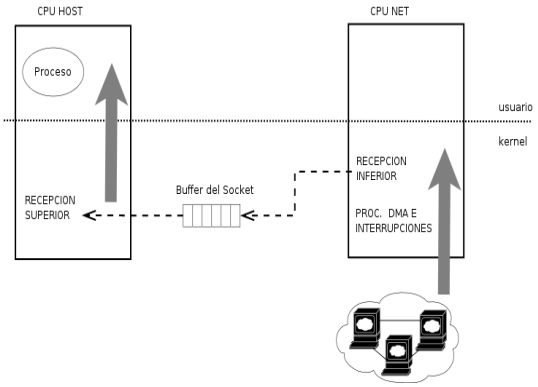

Figura 3: Diagrama del prototipo - Proceso de recepción

Cada CPU-NET ejecuta, en un loop cerrado en contexto del kernel, una secuencia de invocaciones a las funciones correspondientes para avanzar en el procesamiento de red. Con respecto a la recepción, se logra verificar el estado de las NICs e invocar a la softirq de recepción sincrónicamente. La copia final de los datos al espacio de usuario se efectúa en el contexto de la llamada al sistema recv() que realiza el proceso servidor que se ejecuta en la CPU-HOST. Por lo tanto, siempre se realiza el procesamiento de recepción inferior y el procesamiento de las interrupciones en la CPU-NET.

Con respecto a la transmisión, se implementaron dos variantes, dependiendo de cuanto procesa-

miento se desea delegar a la CPU-NET. Se utiliza un buffer circular que implementa el mecanismo de comunicación por memoria compartida entre las CPUs-HOST y las CPUs-NET, por lo tanto se debe consultar constantemente por nuevos requerimientos de transmisión. En la primer variante, la CPU-HOST realiza la transmisión superior e inferior y luego encola el requerimiento para que la CPU-NET termine de efectuar la transmisión. En cambio, la segunda variante delega la transmisión inferior a la CPU-NET, liberando de este trabajo a la CPU-HOST.

En las placas de red se desactivan las interrupciones y se verifica el estado por *polling*. El resto de las interrupciones del sistema son enrutadas a la CPU-HOST usando los mecanismos de enrutamiento ofrecidos por el Controlador Programable Avanzado de Interrupciones de E/S (I/O APIC) [7]. De esta forma, la CPU-NET no debe manejar eventos asincrónicos no relacionados con la red. Sin embargo, la interrupción del timer del APIC local se mantiene habilitada en la CPU-NET, ya que se utiliza para implementar los timers por software del kernel Linux (muy utilizados por TCP), pero además deja abierta la posibilidad de disparar eventos periódicos, como pueden ser evaluar la situación y realizar una reconfiguración automática del subconjunto de CPUs-NET.

Se modificó el kernel Linux 2.6.17.8 para implementar la arquitectura separada de CPU-HOST y CPU-NET en un servidor con dos CPUs. Las pruebas se realizaron sobre máquinas con dos procesadores AMD Opteron de 64 bits, Serie 200 DE 2 GHz (denominadas CPU0 y CPU1). Las máquinas comunican las CPUs con la memoria principal (2 Gbytes de memoria DIMM) a través de interfaces DDR de 144 bits. La comunicación entre ambas CPUs y entre la CPU0 con el chipset se realiza sobre links HyperTransport que soporta una transferencia de datos de hasta 6.4 GB/s. Se instalaron tres NICs gigabit para poder saturar a las CPUs con procesamiento de red, sin embargo, un fenómeno que llamó la atención inmediatamente es que el rendimiento de red agregado de las tres NICs no superaban 1,5 Gbps en las pruebas, aunque las CPUs no llegaban a ocuparse al 100 %, lo que indicaba que el cuello de botella se encontraba en otro componente. Finalmente se descubrió que el problema era bus PCI compartido por las placas.

3.2. Las pruebas de Performance

Para realizar las mediciones combinadas de rendimiento de red y el costo de procesamiento se utilizaron tres herramientas:

- Netperf [1] permite medir el rendimiento de red promedio de un enlace abriendo una conexión TCP entre un cliente y un servidor y enviando datos por un período de tiempo configurable.
- Sar [3] es una herramienta muy completa que permite medir gran cantidad de indicadores de performance en una máquina SMP identificándolos por CPU durante un tiempo determinado y luego ofrece una media de esos datos.
- [2] Oprofile es un profiler estadístico que usa los contadores de performance que ofrecen los procesadores modernos y que permiten obtener información detallada de la ejecución de los procesos, librerías y el propio kernel. Se configura para muestrear determinados eventos, por ejemplo ciclos de CPU ocurridos o fallos de cache L2, y volcar el estado de la CPU, cada una cantidad configurable de muestras obtenidas de un evento dado.

En el documento de la tesis se describe en detalle el conjunto completo de pruebas realizadas (siete en total). Aquí sólo se presentan dos configuraciones que permiten comparar algunos resultados. En la prueba A, se configuró el kernel para lograr afinidad total del procesamiento de dos de las NICs a la CPU0 y el procesamiento de la tercer NIC se asoció a la CPU1. Esta es la configuración que logró mayor eficiencia sin modificar el código

fuente. En la prueba B se utilizó la segunda variante del prototipo implementado. En ambas pruebas se logró un rendimiento de red cercano a 1,5 Gbps utilizando aproximadamente el 70 % de la capacidad de cómputo de la máquina.

Cuadro 1: Comparación de pruebas A y B

medidas\pruebas	A	B
cswch/s	157186,17	99275,49
intr/s	131363	0

Cuadro 2: Perfilamiento del kernel durante la prueba A

muestras	%	símbolo
154270	13.1928	__copy_to_user_ll
153381	13.1167	_spin_lock
94636	8.0930	handle_IRQ_event
58272	4.9833	__do_IRQ
45311	3.8749	schedule
39026	3.3374	tcp_v4_rcv
30196	2.5823	qdisc_restart
26420	2.2594	__switch_to
24781	2.1192	default_idle

Si se comparan las pruebas A y B en el Cuadro 1, se observa que la segunda prueba reduce en más del 30 % la cantidad de cambios de contexto por segundo (cswch/s) y lleva a cero la cantidad de interrupciones de hardware (intr/s).

Los cuadros 2 y 3 muestran los resultados del perfilamiento y permiten estimar, en términos relativos, las funciones que consumieron más tiempo de CPU. El cuadro 2 ofrece un ejemplo claro de la importancia de la intrusión del sistema operativo en el procesamiento de tráfico de red. Aunque que la CPU está ocupada en más del 70 %, las primeras cinco funciones en la lista implementan el manejo genérico de interrupciones, sincronización, copias físicas en memoria principal y planificación de la CPU, asociados al procesamiento de los paquetes de red, mientras que la ejecución real de protocolos de red consume un pequeño porcentaje en comparación a las funciones anteriores. Al perfilar la ejecución del prototipo, se ve en el Cuadro 3 que los costos de sincronización (la función _spin_lock()) bajan notablemente y el subsistema de interrupciones (las funciones handle_IRQ_event() y __do_IRQ()) ya no genera costos importantes.

Cuadro 3: Perfilamiento del kernel durante la prueba B

muestras	%	símbolo
148053	10.3428	__copy_to_user_ll
120776	8.4372	net_rx_action
95700	6.6855	desencolar
68739	4.8020	qdisc_restart
65176	4.5531	tcp_v4_rcv
59659	4.1677	schedule
50669	3.5397	try_to_wake_up
48140	3.3630	tcp_rcv_established
43827	3.0617	kfree
36767	2.5685	__tcp_select_window
30361	2.1210	local_bh_enable
28993	2.0254	eth_type_trans

4. Conclusiones

Resulta razonable afirmar que todas las máquinas conectadas a la red en el futuro cercano estarán construidas en base a algún tipo de arquitectura multicore, donde la cantidad de elementos de procesamiento superarán ampliamente los estándares actuales. Por lo tanto, se han realizado esfuerzos para explotar el paralelismo disponible en las máquinas de propósito general, modificando apropiadamente la implementación de las pilas de protocolos.

Linux sigue un paradigma del tipo procesador por mensaje, ya que el I/O APIC distribuye las interrupciones entre los distintos procesadores de una máquina SMP sin tener en cuenta a qué co-

nexión TCP pertenece el paquete recibido que generó la interrupción. Sin embargo, una de las conclusiones más importantes es que el rendimiento de una conexión TCP es altamente dependiente de la cache, y la división de procesador por paquete no ayuda a mejorar el comportamiento de cache. Otro obstáculo importante en la escalabilidad del subsistema de red del kernel Linux es el mecanismo de sincronización utilizado. El manejo de spinlocks llega a consumir gran parte del tiempo de CPU, y esto empeora significativamente cuando se agregan CPUs al procesamiento de red. Finalmente, se observó que el manejo de interrupciones y la implementación actual de las system calls de red son una fuente importante de overhead, incrementando aun más el tiempo de CPU consumido por la intrusión del sistema operativo.

Aunque tradicionalmente los servicios de red son soportados por sistemas basados en UNIX, recientemente se ha incrementado el interés en crear sistemas operativos diseñados exclusivamente para dedicarse a esta tarea. La tendencia creciente hacia un mundo basado en computación centrada en la red (network centric computing) comenzó a desplazar hacia las operaciones de red el foco de estudio de performance en los sistemas operativos modernos.

Es por esto que se plantea la idea de un kernel activo como el núcleo de un sistema operativo orientado a operaciones de red. El uso asimétrico de una máquina paralela permite destinar uno o varios elementos de procesamiento al kernel, permitiendo que éste sea un componente activo del sistema. Esto permite reducir significativamete los costos asociados a la intrusión del sistema operativo, en particular, de los mecanismos de protección y virtualización. En ésta línea, el prototipo implementado demostró que la separación física de las funciones de red del kernel, con respecto al resto del sistema operativo y los procesos de usuario, ayudó a reducir el overhead generado.

Referencias

[1] Netperf. http://www.netperf.org/netperf/.

[2] Oprofile. http://oprofile.sourceforge.net/.

[3] Sar. http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/admin-primer/s1-bandwidth-rhlspec.html.

[4] V. Anand and B. Hartner. TCP/IP Network Stack Performance in Linux Kernel 2.4 and 2.5. IBM Linux Technology Center.

[5] S. P. Bhattacharya and V. Apte. A Measurement Study of the Linux TCP/IP Stack Performance and Scalability on SMP systems. Indian Institute of Technology, Bombay.

[6] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt. Performance Analysis of System Overheads in TCP/IP Workloads. Advanced Computer Architecture Lab. EECS Department, University of Michigan.

[7] Daniel P. Bonet and Marco Cesati. *Understanding Linux Kernel 3rd Edition*. O'Reilly, 2005.

[8] Jonathan Corbet. Van Jacobson's network channels. http://lwn.net/Articles/169961.

[9] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers 3rd Edition*. Novell Press, 2005.

[10] A. P. Fonng, T. R. Huff, H. H. Hum, J. P. Patwardhan, and Regnier G. J. TCP Performance Re-Visited. Intel Corporation. Department of Computer Science. Duke University.

[11] A. Foong, J. Fung, and D.Ñewell. Improved Linux SMP Scaling: User-directed Processor Affinity. INTEL.

[12] Thomas Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems (Networking Series)*. Charles River Media, 2004.

[13] S. J. Muir. Piglet: An Operating System for Network Appliances.

[14] E.Ñahum, D. Yates, J. Kurose, and D. Towsley. Cache Behavior of Network Protocols. Department of Computer Science. University of Massachusetts.

[15] M. Rangarajan, A. Bohra, K. Banerjee, E. V. Carrera, and R. Bianchini. TCP Servers: Offloading TCP Processing in Internet Servers. Design, Implementation, and Performance.

[16] G. Regnier, S. Markineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D.Ñewell, L. Cline, and A. Foong. TCP Onloading for Data Center Servers. IEEE Computer Society.

[17] D. Schmidt and T. Suda. Measuring the Impact of Alternative Parallel Process Architectures on Communication Subsystem Performance. Department of Information and Computer Science. University of California.

[18] Y. Turner, T. Brencht, G. Regnier, V. Saletore, G. Janakiraman, and B. Lynn. Scalable Networking for Next-Generation Computing Platforms. Proceedings of the Third Annual Workshop on System Area Networks, Madrid, Spain, 2004.