Performance y Escalabilidad del Kernel Linux aplicado a Redes de Alta Velocidad

---

Matías Zabaljáuregui

Trabajo final para obtener el grado de

Licenciado en Informática

de la

Facultad de Informática,
Universidad Nacional de La Plata,
Argentina

Director: Lic. Javier Díaz
Co-director: Ing. Luis Marrone

La Plata, Abril de 2007

A mis viejos, Nelly y Héctor

Agradecimiento

A Javier Díaz, por guiarme en estos primeros pasos.

Indice General

1 Introducción **5**
1.1 Evolución en tecnologías y el crecimiento del tráfico de red 5
1.2 Los Problemas actuales del procesamiento de red 7
1.2.1 La intrusión del sistema operativo 7
1.2.2 Accesos a memoria principal y la inefectividad de la memoria cache 10
1.2.3 Buffering y las copias de datos de aplicaciones 11
1.2.4 Computación de protocolos 11
1.3 El objetivo y la estructura de la tesis 12
1.3.1 El objetivo 13
1.3.2 La estructura 14

2 El hardware involucrado en las comunicaciones y los factores que condicionan la performance **15**
2.1 Introducción 15
2.2 Network Interface Controllers 15
2.3 Tecnología de buses de entrada/salida 17
2.4 Transferencias entre los dispositivos de E/S y la memoria principal 19
2.5 La latencia de la memoria principal y la inefectividad de la cache 21
2.5.1 ¿Por que es un problema la latencia de memoria principal? 22
2.5.2 Accesos a memoria principal de TCP/IP 23
2.5.3 Inefectividad de la memoria cache 24
2.5.4 La regla 1GHz/1Gbps 25

3 El Subsistema de red en el kernel Linux 2.6 **27**
3.1 Introducción 27
3.2 Mecanismos de Linux involucrados en el procesamiento de red . . 28

3.2.1 Interrupciones y funciones diferidas 28
3.2.2 La estructura softnet_data 32
3.2.3 Los socket buffers 33
3.3 El procesamiento de red 33
3.3.1 Procesamiento de conexión 33
3.3.2 Procesamiento de recepción 34
3.3.3 Procesamiento de transmisión 35
3.4 Identificando componentes del procesamiento TCP/IP 35

4 Clasificando los costos del procesamiento de red 38
4.1 Introducción 38
4.2 Operaciones DTO 38
4.2.1 Transferencia de datos entre dominios de protección . . . 38
4.2.2 Application Programming Interfaces 41
4.2.3 Manipulación de datos 43
4.2.4 Transferencia de los datos desde la NIC a la memoria principal utilizando la técnica PIO 43
4.3 Operaciones NDTO 44
4.3.1 Creación y manipulación de buffers de red 44
4.3.2 Operación del sistema operativo 45
4.3.3 Operaciones específicas de los protocolos 45
4.4 Costos DTO versus costos NDTO 45

5 La escalabilidad de TCP/IP en Linux para arquitecturas SMP 48
5.1 Introducción 48
5.2 Multiprocesamiento 49
5.3 Paralelización de protocolos 52
5.3.1 Formas de paralelismo 53
5.3.2 Niveles de paralelismo 54
5.4 El kernel Linux 2.6 en arquitecturas SMP 55
5.4.1 El Controlador programable avanzado de interrupciones de E/S 55
5.5 Problemas de Escalabilidad de TCP/IP en Linux 58
5.5.1 Afinidad de procesos e interrupciones a CPUs 60

6 Reduciendo la intrusión del sistema operativo 61
6.1 Introducción 61
6.2 Intrusión del Sistema Operativo 62
6.3 La evolución de los sistemas operativos en búsqueda de menor intrusión 67
6.3.1 Kernel monolítico 67
6.3.2 El microkernel 69

6.3.3 Kernel verticalmente estructurado 70
6.3.4 El próximo paso: un kernel activo 71

7 Proyectos Relacionados **74**
7.1 Introducción 74
7.2 Evolución del procesamiento de red, las primeras optimizaciones 75
7.3 Propuestas más recientes 80
7.3.1 Mejorando la recepción en el driver/NIC 80
7.3.2 TOE 82
7.3.3 Onloading 83
7.4 Arquitectura de los servicios y la E/S asincrónica 86
7.4.1 Modelos de aplicaciones 86
7.4.2 AIO 87

8 Un prototipo básico y los resultados preliminares **89**
8.1 Introducción 89
8.2 Diseño e implementación del prototipo 89
8.3 Algunas mediciones y resultados preliminares 92
8.3.1 Máquinas de prueba 92
8.3.2 Herramientas de Medición e Instrumentación: Sar, Oprofile y Netperf 93
8.3.3 Las pruebas realizadas y algunos comentarios sobre los primeros resultados 93

9 Conclusiones y trabajo futuro **105**
9.1 Trabajo a futuro 107

Indice de Figuras

1.1 Diferencias relativas en la evolución de las tecnologías [14] 10

3.1 Componentes del procesamiento TCP/IP 36

4.1 Distribución de los costos de procesamiento de TCP/IP 46

5.1 Configuración multi-APIC con dos procesadores [7] 56

6.1 Kernel Monolítico 68
6.2 Kernel Vertical 70
6.3 Kernel Activo 72

8.1 Diagrama del prototipo - Proceso de transmisión 90
8.2 Diagrama del prototipo - Proceso de recepción 91
8.3 Diagrama de bloques de la motherboard utilizada en las pruebas 102

Capítulo 1

Introducción

1.1 Evolución en tecnologías y el crecimiento del tráfico de red

La combinación de nuevas tecnologías de redes de alta velocidad (con capacidades de transmisión en el orden de Gigabits por segundo, como Infiniband y Ethernet 10Gbps) y la capacidad de cómputo en los procesadores de última generación han impulsado el nacimiento de nuevos paradigmas de procesamiento paralelo y distribuido. Sin embargo, en esta evolución, el diseño de los sistemas operativos de propósito general y las arquitecturas de hardware más utilizadas en servidores, estaciones de trabajo y dispositivos de red han comenzado a dar señales de limitaciones con respecto a la escalabilidad.

El crecimiento exponencial en las capacidades de transmisión provistas por los nuevos estándares de red han superado ampliamente, en términos relativos, el incremento en el poder de cómputo de la tecnología actual de computadoras. Desde el año 1995 hasta 2002, el estandard IEEE Ethernet ha evolucionado desde una velocidad máxima de 100 Mbps a 10 Gbps, un incremento de dos órdenes de magnitud. Sin embargo, en el mismo período la Ley de Moore indica una mejora de hasta 25 veces en la capacidad de los procesadores, resultando en una clara diferencia relativa en la evolución de estas tecnologías. Las soluciones de capas inferiores de la pila TCP/IP ofrecen capacidades de transmisión de datos lo suficientemente rápidas y a costos tan razonables que están dejando de ser de ser sólo tecnologías de comunicaciones para ser usada también como tecnologías de backplane (ya fueron planificadas futuras versiones de Ethernet a velocidades de 40 y 100 Gbps).

Internet, y las aplicaciones que ésta posibilita, han transformado a las arquitecturas de los Data Centers, desde un modelo sencillo de único servidor a

un modelo multi-tier en el cual servidores web, servidores de aplicación, bases de datos y servidores de almacenamiento trabajan juntos para responder a cada requerimiento de un cliente. Como resultado, la performance general de las aplicaciones se ha vuelto más dependiente de la eficiencia en las comunicaciones.

El Protocolo de Control de transmisión (Transmission Control Protocol o TCP) es el protocolo más usado para comunicaciones de datos confiables tanto en Internet como en redes locales. TCP es la capa de transporte en la pila de protocolos TCP/IP y provee la entrega ordenada y garantizada de los datos a las aplicaciones corriendo en cada uno de los extremos. Éste protocolo suele usarse junto con el Protocolo de Internet (Internet Protocol o IP) que implementa la capa de red de la pila y provee servicios de enrutamiento entre los nodos origen y destino. A estos dos protocolos se los suele denominar colectivamente TCP/IP. Utilizados por primera vez a mediados de la década de 1980, TCP/IP fue creciendo en popularidad con la evolución de Internet y las aplicaciones relacionadas. Dentro de los data centers y las redes locales corporativas, TCP/IP se suele transmitir sobre Ethernet, que provee, entre otras cosas, servicios de tramas a los protocolos superiores. No sólo es el protocolo más utilizado en redes LAN, también está siendo considerado para redes MAN y WAN.

Ya existen nuevas formas de aprovechar las últimas tecnologías en telecomunicaciones, tanto en entornos locales como en redes distribuidas geográficamente. El tráfico de red se está incrementando dramáticamente debido a la confluencia de diversos factores:

- Los negocios están usando las redes para acceder a recursos como el almacenamiento corporativo, que antes se conectaba directamente.
- Los Web Services y la WWW han movido una creciente actividad de negocios a un paradigma basado en redes.
- El pasaje de mensajes se está volviendo la forma principal de compartir datos y servicios.
- Los clusters de computadoras se están volviendo la forma más común de crear recursos de cómputo de alto rendimiento.
- El volumen de datos ha crecido dramáticamente debido al uso creciente de recursos de audio y video, y adquisición de datos en tiempo real (como las redes de sensores y el tracking de tags RFID).
- El desarrollo de la e-ciencia y las tecnologías de Grid Computing desplazarán terabytes de datos por día a través de las redes.

1.2 Los Problemas actuales del procesamiento de red

El salto repentino en las velocidades de Ethernet desde los 100 Mbps a los 10 Gbps requiere que el procesamiento de TCP/IP en los servidores escalen proporcionalmente para que las aplicaciones intensivas en el uso de la red puedan beneficiarse efectivamente de los niveles crecientes de ancho de banda disponible. Es decir, el desafío planteado es utilizar el crecimiento del ancho de banda de red para producir una ganancia proporcional en el rendimiento de la aplicación.

Durante años, mientras crecía la disponibilidad de ancho de banda y mejoraba la performance del hardware de red, el poder de cómputo ofrecido por las máquinas conectadas a la red era suficiente para realizar el procesamiento necesario para enviar y recibir paquetes sin interferir en la ejecución de los procesos de usuario. Sin embargo, las arquitecturas de hardware y el diseño de software del subsistema de red de las maquinas actuales han comenzado a dar señales de ciertas limitaciones con respecto a la escalabilidad en el procesamiento de red.

A continuación se mencionan los factores que han sido identificados como fuente de overhead importante en el subsistema de red de los sistemas operativos actuales.

1.2.1 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 overhead introducido por éste al realizar sus funciones principales de virtualización y protección de los recursos de la máquina. Las interrupciones por hardware y las *system calls* implementadas como traps son un ejemplo de intrusión por mecanismo (intrusión relacionada con la implementación de las funciones del sistema operativo). Por ejemplo, en los modelos tradicionales de drivers para interfaces de red, el procesador es interrumpido por cada paquete recibido. Sin embargo, las interfaces de alta velocidad pueden recibir miles de paquetes por segundo, generando miles de interrupciones (y, en consecuencia, miles de intercambios de tareas) por segundo. A esta frecuencia, las interrupciones por hardware y las system calls que se implementan como interrupciones programadas se vuelven un mecanismo demasiado costoso. Además de quitarle ciclos de procesador a la aplicación, las interrupciones asincrónicas y cambios de contextos frecuentes, también causan efectos indirectos como la polución de la memoria cache y del *Translation Lookaside Buffer*[1] (TLB), lo que incrementa aun mas el overhead debido al procesamiento de red.

Los sistemas multiprocesadores tienen más probabilidades de sufrir intrusión por mecanismo, ya que la mayoría de los sistemas operativos de propósito general

[1]Los procesadores de la familia x86 incluyen este tipo de cache para acelerar las traducciones de direcciones lineales a direcciones físicas.

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. Teniendo en cuenta que se prevé un incremento en la disponibilidad de máquinas con arquitecturas Chip Multiprocessing (CMP) y Symmetric Multiprocessing (SMP), el problema planteado sólo se agudizará si no se realizan las modificaciones necesarias. Se ha demostrado que los actuales sistemas operativos de propósito general no hacen un uso eficiente del hardware SMP/CMP en el procesamiento de red [11, 4, 20]. Como resultado, se genera un overhead notable en el tratamiento de las operaciones de entrada/salida que amenaza con impedir el completo aprovechamiento del ancho de banda disponible en los próximos años.

Overhead del software

En el modelo cliente-servidor, el diseño y la arquitectura del proceso servidor juega un rol importante para determinar la performance y la escalabilidad del sistema completo. Para permitir el procesamiento concurrente de requerimientos de múltiples clientes se disponen de dos alternativas principales: Las aplicaciones de servicios típicamente son multi-threaded o multi-process. El modelo multi-process involucra frecuentes cambios de contextos entre varios procesos y el uso de primitivas IPC. En el caso multi-threaded, el contexto es más liviano y la comunicación puede hacerse por memoria compartida, sin embargo, en servidores que atienden a miles de clientes simultáneamente el scheduling de threads puede generar un overhead importante.

Por otro lado, la pila TCP/IP ha sido tradicionalmente implementada por software como parte del kernel de los sistemas operativos. TCP/IP interactúa con las interfaces de red (NICs, Network Interface Controllers) a través de un driver de dispositivo y se expone a las aplicaciones a través de la interfaz tradicional de sockets. El procesamiento TCP/IP comienza en el hardware de la NIC y se extiende a través de toda la pila hasta la capa de aplicación. El sistema operativo es el responsable de la alocación y administración de los recursos compartidos del sistema. El acceso a la NIC, disco y otros dispositivos de E/S es protegido bajo el control del sistema operativo, el cual también es responsable por la planificación de la CPU entre varios procesos que compiten. Sin embargo, las políticas en las que se basan el scheduling, la alocación de memoria principal y otras funcionalidades suelen estar diseñadas para proveer fairness en un sistema de tiempo compartido, en lugar de maximizar la eficiencia en el movimiento de datos.

Las system calls representan la interfaz entre el servidor y el kernel, y proveen un mecanismo para que las aplicaciones requieran servicios del sistema operativo. Una system call típicamente es ejecutada como un trap (interrupción

programada por software) la cual causa que la ejecución salte a un punto de entrada fijo en el kernel. Una vez dentro del contexto del kernel, operaciones privilegiadas (como acceso a los dispositivos) son realizadas por el sistema operativo en nombre de la aplicación que lo requiere. Pero antes de que la ejecución cambie desde el la aplicación al kernel, el estado de la aplicación es salvado, y luego restaurado justo antes de que la llamada retorne, por lo tanto las system calls son un mecanismo costoso. Las operaciones con sockets, que permiten a las aplicaciones realizar operaciones de red, se implementan como system calls y los datos deben ser copiados desde los buffers de la aplicación a los buffers del kernel y viceversa. Esta copia agrega costos significativos a todo el proceso.

Interrupciones del hardware

Un servidor intentando transmitir/recibir a velocidades de los enlaces de alta performance (10 Gbps) debe manejar paquetes de red que se suceden uno detrás de otro a un intervalo de tiempo menor que la suma de las latencias producidas por accesos a memoria principal y computaciones correspondientes. Para complicar las cosas, la mayoría del tiempo de utilización de la CPU para procesar paquetes no es productivo. La gran latencia de acceso a memoria principal fuerza a la CPU a paralizarse mientras espera que se completen los requerimientos de datos en RAM (motivo por el cual surgieron las variantes de Simultaneous MultiThreading, como la tecnologia HyperThreading de Intel).

El procesamiento del protocolo TCP/IP requiere una cantidad significativa de recursos del sistema y compite con el tiempo de procesamiento de las aplicaciones. Este problema se vuelve más crítico con cargas altas de trabajo porque:

- La llegada de un paquete resulta en una interrupción asincrónica lo que implica una interrupción de la ejecución actual, sin importar si el servidor dispone de recursos suficientes para procesar completamente el paquete recién llegado.
- Cuando un paquete finalmente alcanza la cola del socket, puede ser desechado porque no hay recursos suficientes disponibles para completar su procesamiento. El descarte del paquete ocurre después de que una cantidad considerable de recursos del sistema hayan sido gastadas en el procesamiento del paquete.

Las condiciones anteriores pueden causar que el sistema entre en un estado conocido como *receive livelock* (explicado en el capítulo 6). En este estado, el sistema consume recursos considerables procesando paquetes entrantes sólo para descartarlos más tarde por la falta de recursos necesarios para que la aplicación procese los datos.

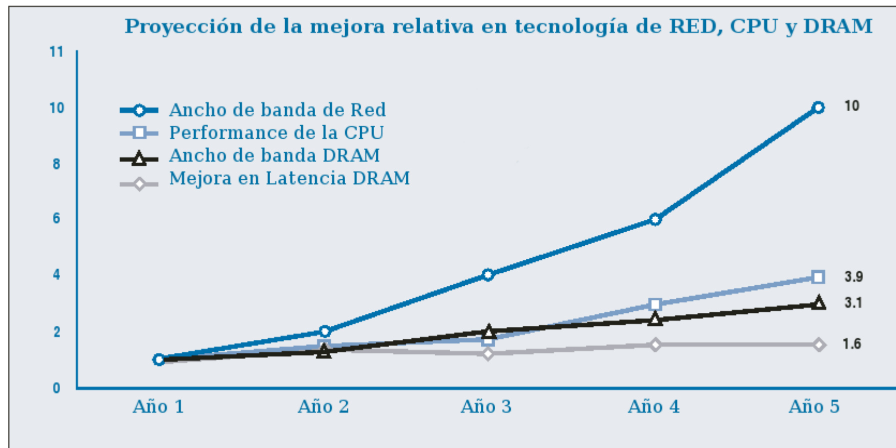

Figura 1.1: Diferencias relativas en la evolución de las tecnologías [14]

1.2.2 Accesos a memoria principal y la inefectividad de la memoria cache

Otro gran desafío para el procesamiento TCP/IP es la latencia de la memoria principal y la escasa localidad de cache en el procesamiento de red. La memoria principal representa un importante cuello de botella, agregando latencia y generando ciclos ociosos de la CPU. Además, TCP/IP es lo que se conoce como una *flow application*, esto significa que no hay una forma natural de usar la memoria cache para explotar localidades espaciales o temporales que reduzca la latencia de los accesos a memoria. La memoria principal y cache de los servidores están optimizadas para una ejecución eficiente de un amplio conjunto de aplicaciones, pero el software TCP/IP generalmente exhibe una localidad de cache pobre, principalmente, porque maneja grandes cantidades de información de control y datos que está entrando o saliendo del sistema a través del hardware de red.

Una vez que la NIC recibe paquetes y los coloca en memoria, el software TCP/IP debe examinar la información de control que incluye el descriptor y encabezamiento para cada paquete. Además los datos de la aplicación (el payload del paquete) deben ser copiados a su destino final. Como el paquete fue colocado en memoria justo antes, el acceso a estos datos siempre causarán fallos de cache. Es decir, las cargas intensivas en E/S de red no demuestran localidad de las referencias, causando numerosos fallos de cache y por lo tanto gran cantidad de accesos a memoria principal.

Por ejemplo, el código TCP/IP de Linux puede provocar que la CPU acceda hasta cinco veces a la memoria RAM para procesar un único paquete [14] y cada acceso implica una gran cantidad de ciclos de CPU desperdiciados debido a las

diferencias de velocidad entre la CPU y el acceso a memoria (esto se estudia en el capítulo 2).

1.2.3 Buffering y las copias de datos de aplicaciones

Si la aplicación no ha indicado ningún destino en el momento en que se reciben datos, la pila TCP/IP copia los datos entrantes en buffers internos. Luego, se copiarán los datos nuevamente en el buffer de la aplicación cuando éstos estén disponibles. Por lo tanto, dependiendo del escenario, puede haber una o dos copias de datos entrantes.

Cuando se transmiten datos, existe una oportunidad de evitar una copia transfiriendo los datos directamente desde los buffers de la aplicación a la NIC, usando Direct Memory Access (DMA). La transmisión sin copias en memoria, universalmente conocida como *zero copy* ha sido lograda en algunas implementaciones, por ejemplo, incluyendo un servidor web en el kernel Linux, utilizando la system call sendfile(), a través del mecanismo zero-copy de Solaris, etc.

Las implementaciones basadas en sockets BSD tradicionales, sin embargo, todavía requieren en el caso general una copia entre los buffers de la aplicación y el kernel. Estas copias de datos son costosas, especialmente cuando alguno de los buffers de origen o destino no están en memoria cache. Por este motivo, se están proponiendo interfaces alternativas de sockets (como se menciona en el capítulo 7).

1.2.4 Computación de protocolos

Las pilas de protocolos actuales se basan en algoritmos y código creados en los primeros años del desarrollo de las redes de datos, con funcionalidad agregada a lo largo del tiempo. Esto resultó en diseños e implementaciones no del todo eficientes, incrementando el tiempo y recursos necesarios para el procesamiento de cada paquete.

Numerosas publicaciones plantean que las pilas de protocolos utilizadas en los actuales sistemas operativos han quedado obsoletas [10, 6, 14, 12, 8, 26, 25]. Los problemas estructurales que se presentan en estas implementaciones fueron heredados, en gran medida, por decisiones de diseño volcadas en las primeras versiones de TCP/IP, pensadas para redes soportadas por medios físicos mucho menos confiables y con velocidades de transmisión que representan sólo una pequeña fracción de los estándares actuales. Además, aunque se han agregado nuevas funcionalidades a los protocolos originales, en general, estas modificaciones al código se han implementado como adiciones posteriores al diseño original, resultando en una mayor latencia en el procesamiento. Todo indica que es tiempo de un nuevo diseño.

1.3 El objetivo y la estructura de la tesis

Mientras que los esfuerzos de aceleración de TCP/IP se enfocan en lograr velocidades de comunicación de 10 Gbps, mejorar la eficiencia en el procesamiento de red es igualmente importante. Algunos estudios han demostrado que el overhead de procesamiento de red puede ser tan alto como el 77 % del uso total de CPU en servidores Web con arquitecturas SMP [8].

Existe una regla empírica generalmente aceptada que indica que se necesita 1 GHz de procesamiento de CPU para manejar 1 Gbps de tráfico de red (esta relación empeora a medida que aumenta la frecuencia de la CPU) [10]. En servidores con conexión multigigabit y una velocidad de reloj típica de procesador de 3 GHz, los ciclos de CPU disponibles para las aplicaciones se ven limitados por el procesamiento de los paquetes de red.

Para evitar los serios problemas de escalabilidad en el futuro, debe mejorarse drásticamente la eficiencia del procesamiento TCP/IP teniendo en cuenta que a medida que diferentes tipos de aplicaciones (algunas de rendimiento intensivo y otras altamente sensibles a la latencia) comienzan a utilizar TCP/IP, el proceso de acelerar y escalar la pila TCP/IP requiere una aproximación balanceada. Las plataformas de cómputo del futuro requerirán procesamiento de protocolos y paquetes rápido y de bajo overhead para alimentar a las interfaces emergentes de Entrada/Salida (E/S) de alta velocidad.

Las plataformas del futuro también necesitarán mayor concurrencia de E/S para aprovechar completamente la ventaja de los nuevos enlaces de comunicaciones. Se hace necesario tener varias operaciones de E/S ejecutándose concurrentemente para utilizar efectivamente el ancho de banda del enlace. Sin embargo, las aproximaciones convencionales de multithreading del software actual no escala lo suficiente, agregando overhead por el scheduling y la sincronización[25].

Existen proyectos, originados tanto en entornos académicos como en la industria, para afrontar los problemas mencionados. Las soluciones que se destacan han propuesto modificaciones en el codiseño de hardware/software, atacando cada uno de los puntos débiles del modelo anterior: reducción de accesos a RAM buscando operaciones zero-copy, implementación por hardware (o firmware) de cierta funcionalidad en las interfaces de red, reducción de interrupciones de hardware o su reemplazo por técnicas de polling, optimizaciones de las implementaciones de TCP/IP o su implementación en espacio de usuario, capas de sockets con semántica asincrónica para evitar el uso del multithreading, mejoras en el uso de memoria cache, y alternativas de aproximaciones a la paralelización de protocolos de red, entre otros.

1.3.1 El objetivo

El incremento en la disponibilidad de las actuales máquinas SMP/CMP ofrece la oportunidad de utilizar y distribuir el poder de cómputo de nuevas formas. Queda abierto el desafío para los desarrolladores de software de encontrar la mejor forma de aprovechar estos recursos.

En la evolución de los sistemas operativos, desde el antiguo diseño de kernel monolítico (pasando por microkernels, exokernels o kernels verticales, y otras variantes) se ha llegado a lo que se denominó como kernel activo. Este nuevo diseño, presentado por Steve J. Muir en el año 2001 [19], plantea un kernel especializado en procesamiento de red, motivado principalmente por la poca eficiencia con que los sistemas operativos de propósito general realizan este tipo de tareas en máquinas SMP. Su trabajo propone dedicar, de manera exclusiva, uno o más procesadores a tareas específicas del kernel, permitiendo que las aplicaciones de usuario se ejecuten, con la menor cantidad de intrusión por parte del sistema operativo, en el resto de los procesadores. De esta forma, el kernel deja de ser un proveedor pasivo de servicios para convertirse en un proceso activo del sistema, y la separación física de procesos de usuario con respecto a threads del kernel evita incurrir en el overhead necesario para implementar el modelo *usuario/kernel*.

Esta tesis pretende continuar en esta línea de investigación, estudiando la forma de adaptar el kernel Linux a los altos requerimientos de procesamiento de red a los que estarán expuestos los servidores, dispositivos de red y estaciones de trabajo en un futuro cercano. Para lograr este objetivo, se estudia en profundidad el codiseño hardware/software del subsistema de red en un servidor con placas de red gigabit Ethernet, arquitectura SMP y el kernel Linux 2.6. También se hacen pruebas de performance y profiling del modelo actual de procesamiento de paquetes para compararlo con el prototipo propuesto más adelante.

Complementando este trabajo, se analizan los problemas mencionados anteriormente y algunas soluciones propuestas por diversos grupos académicos y del ámbito industrial. En éste último entorno, el caso que destaca del resto es la nueva tecnología que está siendo desarrollada por Intel, denominada *I/O Acceleration Technology* [14, 15], que utiliza varios de los conceptos mencionados en los trabajos citados anteriormente.

Finalmente, se propone una modificación al subsistema de red del kernel Linux 2.6 sobre máquinas SMP, basándose en las ideas más recientes relacionadas con la evolución de los kernels. De esta forma, se logra evaluar el efecto de hacer un uso asimétrico de los procesadores en una máquina SMP, asignando un procesador, o un subconjunto de ellos, de manera exclusiva al procesamiento de red.

1.3.2 La estructura

A continuación se mencionan los temas abordados en el resto de los capítulos de la tesis. En el capítulo 2 se estudia el hardware involucrado en el proceso de transmisión y recepción de paquetes TCP/IP sobre Ethernet, haciendo hincapié en los factores que condicionan la eficiencia. El capítulo 3 introduce los mecanismos del kernel Linux directamente asociados al procesamiento de red y resume la secuencia de eventos que suceden en el kernel cuando se envían y reciben paquetes. En el capítulo 4 se identifican y clasifican las distintas fuentes de overhead generado en el procesamiento de red y se mencionan algunas técnicas que han intentado mejorar ese comportamiento. El capítulo 5 analiza las alternativas de paralelización de protocolos de red e identifica la aproximación seguida por el kernel Linux y los problemas de escalabilidad que ésta implica. El capítulo 6 explica el concepto de intrusión del sistema operativo y compara las diferentes formas de diseño de un kernel con respecto a la eficiencia en el procesamiento de entrada/salida. El capítulo 7 menciona las optimizaciones que se han realizado a la pila TCP/IP y los proyectos más recientes relacionados con este trabajo. En el capítulo 8 presenta la modificación al kernel Linux implementada para este trabajo y se ofrecen los resultados de un conjunto de pruebas comparativas. Finalmente, la conclusión de esta investigación y las ideas para un trabajo a futuro se mencionan en el capítulo 9.

Capítulo 2

El hardware involucrado en las comunicaciones y los factores que condicionan la performance

2.1 Introducción

El propósito de este capítulo es el de hacer una introducción conceptual al hardware de comunicaciones utilizado por una gran cantidad de servidores, estaciones de trabajo y algunos dispositivos de red. Se describe el hardware relacionado con la transmisión de datos en un bus Ethernet y su relación con el bus de E/S, memoria principal y memoria cache, haciendo hincapié en las implicaciones de performance.

Se pospone el estudio del rol de la CPU en el procesamiento de red, ya que se dedica un capítulo entero a los temas relacionados con la paralelización de protocolos en arquitecturas SMP/CMP, que son el objetivo principal de este trabajo.

2.2 Network Interface Controllers

Las NICs del tipo Gigabit Ethernet suelen ser dispositivos altamente integrados y de alta performance para velocidades de transmisión de 10 Mbps, 100 Mbps y 1000 Mbps, generalmente conectados a un bus PCI o PCI-X. Proveen una interfaz con la CPU basada en registros de estado, comandos on-chip y un área de memoria RAM compartida, definida durante la inicialización del hardware, que contiene buffers de datos de transmisión y recepción, y los descriptores de esos buffers.

El software aloca los buffers e inicializa los descriptores que contienen el estado y los punteros a esos buffers. Luego se le da al hardware el control de la cola de buffers de recepción, los cuales almacenarán los datos recibidos desde la red. Para la transmisión, el software mantiene una cola de buffers. Cuando el driver quiere transmitir un buffer, emite el comando correspondiente al hardware. Éste controlará el buffer hasta que los datos sean cargados en la memoria fifo de la NIC (explicado más adelante) para su transmisión y luego será liberado para ser posteriormente desalocado o reciclado por el software.

Las interfaces incluyen un buffer fifo que consiste en memoria on-chip que provee un espacio de almacenamiento temporal para tramas que son recibidas o serán transmitidas por el controlador Ethernet. Además poseen un controlador DMA que maneja las transferencias de datos y descriptores entre la memoria principal de la computadora y la memoria fifo en la NIC. En la recepción, el controlador DMA transfiere los datos almacenados en el buffer fifo de recepción al buffer de recepción en memoria principal. La NIC se encarga de leer los descriptores de memoria y volver a escribirlos en memoria una vez actualizados con la información de control correspondiente a los datos recibidos. En el camino de transmisión, el controlador DMA transfiere los datos almacenados en los buffers en la memoria principal al buffer fifo de transmisión de la NIC. También realiza las operaciones necesarias con los descriptores en memoria de los buffers de datos utilizados.

Este tipo de hardware se suele basar en el mecanismo de interrupciones para comunicarse con el driver encargado de manejarlo. En las placas más modernas, se usan técnicas para minimizar el número de interrupciones necesario, por ejemplo, recibiendo y almacenando varias tramas antes de interrumpir al procesador, en lugar de interrumpirlo por cada trama recibida. Más adelante se detalla esta técnica conocida como *interrupt mitigation*.

Por último, y por motivos de performance, en los últimos años se han comenzado a trasladar al hardware de la NIC cierta funcionalidad que tradicionalmente se implementaba por software. El *Checksum Offloading* es la característica que permite desentender al software del cálculo del checksum IP, TCP y UDP. En la transmisión, se pueden calcular hasta dos checksums para un paquete Ethernet, típicamente el checksum IP y el TCP o UDP. En la recepción, el hardware reconoce el tipo del paquete y realiza el cálculo del checksum y el chequeo de errores automáticamente. La información de checksum y errores es provista al software a través de los descriptores de recepción, aunque también existe la posibilidad de filtrar directamente las tramas cuando la verificación de checksum no es válida, evitando procesamiento posterior de la CPU sobre una trama que finalmente será descartada.

Los controladores Ethernet también pueden implementar la capacidad de segmentación TCP para la transmisión, normalmente conocido como TCP Segmentation Offloading (TSO) o Large Segmentation Offloading (LSO). El driver

del dispositivo puede enviar el mensaje completo, como lo recibió de las capas superiores, y el controlador realiza la tarea de segmentar el paquete. Esto reduce la utilización de CPU para la transmisión y es una característica común en controladores Gigabit Ethernet.

2.3 Tecnología de buses de entrada/salida

La ubicuidad de las comunicaciones basadas en TCP/IP ha llevado al subsistema de red, de ser una característica opcional de los sistemas operativos, a tener un rol central en el funcionamiento del mismo. Se espera que en un futuro las operaciones de red sean casi tan frecuentes como el resto de las operaciones básicas de una computadora.

Sin embargo, las placas de red en las arquitecturas más utilizadas siguen siendo consideradas como dispositivos periféricos genéricos conectados a los buses de Entrada/Salida estándares. Dado que las prestaciones de estos buses evolucionan a una velocidad relativamente lenta, su performance suele quedar por debajo de la performance de la CPU, hardware de red y el ancho de banda disponible. Por ejemplo, es posible comprar una placa Ethernet 10 Gbps full-duplex, con una capacidad de transmisión bidireccional de 20 Gbps. Pero el máximo rendimiento teórico se limita a 8 Gbps, debido al bus PCI-X (de 64 bit a 133Mhz) usado por esta placa. Al igual que el cuello de botella de las conexiones WAN suele estar en la *última milla*, en una LAN de alta velocidad el cuello de botella se encuentra en los últimos milímetros que separan a la NIC de la CPU y memoria principal.

Mientras que los estándares PCI-X 2.0 y PCI Express están comenzando a solucionar esta disparidad de ancho de banda, no reducen significativamente la latencia de comunicación entre la CPU y la NIC, que actualmente es de miles de ciclos de CPU.

El bus PCI

En 1993 Intel lanzó el bus PCI (Peripheral Component Interconnect), el cual rápidamente se convirtió en el bus estándar para servidores y máquinas de escritorio. Sin embargo, ya a mediados de la década de 1990, surgieron dispositivos que consumían más ancho de banda del que PCI podía ofrecer (por ejemplo, las placas de video). Con velocidades de reloj de 33MHz y un ancho de datos de 32 bits, el ancho de banda teórico total de PCI es de 132 megabytes por segundo. Aunque el estándar original incluía la posibilidad de un ancho de datos de 64 bits y revisiones posteriores (PCI 2.2) incrementaron la velocidad del bus a 66 MHz, logrando 532 megabytes por segundo con dispositivos de 64 bits, estas configuraciones no representaron el caso general en un principio.

PCI-X

PCI-X 1.0 (Peripheral Component Interconnect Extended) fue desarrollado en conjunto por IBM, HP, y Compaq, y surge en 1998 con la intención de superar los límites de PCI para soportar las nuevas tecnologías de gran ancho de banda, tales como Gigabit Ethernet, Fibre Channel, InfiniBand, SCSI e iSCSI. Esencialmente es una versión más rápida de su antecesor, ya que comparten muchos aspectos de diseño e implementación física. PCI-X duplica la frecuencia lograda por PCI 2.2, obteniendo un bus de 64 bits que transmite datos a 133 MHz. Es decir, se alcanza una taza de transferencia límite de 1,06 gigabytes por segundo para dispositivos de 64 bits.

PCI-X 2.0 es una nueva versión aun más rápida que añade dos nuevas velocidades: PCI-X 266 y PCI-X 533, permitiendo así hasta 4.26 gigabytes por segundo de capacidad de transferencia (32 veces más rápido que la primera generación de PCI). Otra característica de PCI-X 2.0 es su mayor fiabilidad. Se ha añadido soporte de ECC tanto para las cabeceras como para los datos, ofreciendo así recuperación de errores de un único bit, y detección de errores en dos bits.

PCI Express

PCI Express (denominado a veces por su nombre clave 3GIO, por "tercera generación de E/S") es el sucesor de la tecnología PCI. Al contrario que sus predecesores paralelos, PCI Express es un sistema de interconexión serie punto a punto capaz de ofrecer transferencias a altas velocidades. Esto refleja una tendencia en la industria de reemplazar los antiguos buses compartidos paralelos con buses seriales de alto rendimiento como Serial ATA (SATA), Serial-Attached SCSI (SAS), HyperTransport, USB y FireWire.

Un link PCI Express consiste en dos canales simplex, cada uno implementado como un par de transmisión y un par de recepción para una comunicación bidireccional. El ancho de banda de una conexión PCI Express puede ser aumentado agregando pares entre los dispositivos, desde 2,5 gigabits por segundo para la implementación 1X, hasta 80 gigabits por segundo para el actual máximo de 32X. La notación 1X y 32X se refiere al número de links que conforman la conexión.

PCI Express también incluye características novedosas, tales como gestión de energía, conexión y desconexión en caliente de dispositivos (como USB), y la capacidad de manejar transferencias de datos punto a punto. Esto último es importante porque permite a PCI Express mover datos entre dos dispositivos compatibles sin necesidad de que éstos pasen primero a través del hardware del host.

En enero del 2007 se anunció la especificación base de PCI Express 2.0, la

cual duplica la capacidad de transmisión pasando de 2,5 a 5 gigabits por segundo para un link 1X.

2.4 Transferencias entre los dispositivos de E/S y la memoria principal

Siendo una de las operaciones fundamentales relacionadas con la performance de red, el movimiento de los datos entre la memoria incorporada en la interfaz de red y la memoria principal de la máquina es uno de los temas que ha merecido gran atención por parte de los diseñadores.

Las técnicas más utilizadas son Direct Memory Access (DMA) y Programmed Input/Output (PIO). DMA permite que un adaptador de E/S transfiera los datos directamente desde o hacia la memoria principal sin que se involucre la CPU. PIO requiere que el procesador transfiera palabras individuales (típicamente 32 bits) entre la memoria principal y el adaptador de E/S en un loop programado. Si bien la gran mayoría del hardware moderno se basa en la técnica DMA para esta transferencia, aun se continúa el estudio de combinaciones de técnicas alternativas.

A continuación se realiza una evaluación comparativa de estas técnicas para el caso específico de transferencias de datos de red entre la NIC y la memoria principal, teniendo en cuenta la relación con los distintos componentes que participan de la transferencia.

Con respecto a la CPU

Con DMA, la NIC directamente lee y escribe la memoria principal sin intervención de la CPU. La CPU sólo debe indicarle a la NIC en qué direcciones de memoria puede leer y escribir (esto lo hace a través de los descriptores mencionados anteriormente). Con esta técnica, no sería necesario almacenar tramas en la memoria de la NIC, a excepción de algunos bytes que se almacenan en el buffer fifo, que hace de intermediario entre el enlace de red y el bus del sistema[1]. DMA puede trabajar en paralelo con una computación en la CPU, lo cual significa que la aplicación pierde menos ciclos de CPU por la comunicación. Sin embargo, esta paralelización depende tanto del ancho de banda de memoria disponible como de la tasa de aciertos de cache de la aplicación, ya que la contención por el acceso a memoria puede paralizar al procesador en períodos de mucho tráfico DMA.

En la técnica PIO, la CPU es responsable de mover los datos entre la NIC y la memoria principal. Para enviar una trama, la CPU entra en un loop donde

[1] Las placas modernas incluyen, por cuestiones de performance, varios kilobytes de memoria fifo aunque utilicen DMA.

primero lee una palabra de memoria y luego la escribe en la NIC. Para recibir una trama, la CPU lee palabras de la NIC y las escribe en memoria, por lo tanto la CPU está ocupada durante las transferencias desde o hacia un dispositivo.

Con respecto a los buses de E/S

La mayoría de los buses de E/S dependen del uso de transferencias de bloques de datos de un tamaño relativamente grande para lograr un alto rendimiento. Con DMA, generalmente es posible transferir grandes bloques de datos en una única transacción del bus, logrando velocidades de transferencia cercanas al límite de la capacidad del bus de E/S y de memoria principal. Claramente, cuando la CPU está involucrada en el movimiento de los datos, las unidades de transferencia son palabras. Por lo tanto, DMA es más eficiente en el uso del bus. Además DMA requiere sólo una transferencia por palabra sobre el bus de memoria, mientras que PIO requiere dos: una lectura a registro y una escritura a memoria. Por lo que el uso de DMA reduce el tráfico en el bus de memoria. Con PIO, sólo una fracción del ancho de banda máximo es alcanzado.

Con respecto a la memoria principal

Tramas diferentes se ubican en buffers separados, sin embargo, una única trama podría ser distribuida entre múltiples buffers. Esta característica es conocida como *scatter-read* y en la práctica se utiliza cuando el tamaño máximo de trama es tan grande que no resulta eficiente alocar todos los buffers de ese tamaño. Luego, el sistema operativo utiliza algún mecanismo para asociar todos los buffers que conforman una única trama. Por otro lado, cuando el host tiene una trama para transmitir, coloca un puntero al buffer que contiene la trama en la lista de descriptores de transmisión. Los dispositivos que soportan operaciones de tipo *gather-write* permiten que la trama esté fragmentada en múltiples buffers en memoria.

Las tramas a ser enviadas son construidas en etapas, con información agregada por cada capa de la pila de protocolos. Un mensaje que termina de recorrer la pila de protocolos y está listo para ser transmitido generalmente consiste en un buffer que contiene la suma de encabezados de cada protocolo y otro buffer separado que contiene los datos a ser enviados. La combinación *scatter-gather* permite transferencias DMA desde y hacia buffers físicos de memoria no contigua, lo cual ayuda a evitar tener que alocar grandes espacios físicos de memoria y realizar copias físicas de los fragmentos de una trama. En PIO, el scatter-gather debe ser implementado por software, con el overhead que esto significa.

Sin embargo, PIO presenta una ventaja importante con respecto a los accesos a memoria principal. Las computaciones que ocurren en el kernel, como el checksum, pueden ser integradas con el movimiento de datos de PIO, ahorrando

un acceso a memoria (explicado en detalle más adelante). Esta técnica se implementó en el kernel Linux con tanto éxito, que puso en duda la necesidad del checksum integrado en el hardware[4].

Con respecto a la memoria cache

Con PIO, luego de un movimiento de datos por software desde la NIC a la memoria principal, los datos estarán en memoria cache. Esto puede resultar en una reducción del tráfico a memoria principal si los datos son accedidos posteriormente y aun permanecen válidos en la cache.

Por otro lado el uso de DMA podría resultar en una transferencia de datos incorrectos si no se tienen en cuenta las inconsistencias entre la cache y la memoria principal. La operación DMA obtendrá datos antiguos de la memoria principal si los datos correctos sólo aparecen en la cache (sólo en sistemas con cache *write-back*). Se debe proveer la consistencia volcando los datos de la cache en la memoria principal antes de empezar la transferencia DMA. Un problema similar ocurre al recibir datos, aunque puede ser evitado con una invalidación de determinadas líneas de cache. El volcado y la invalidación de cache son operaciones costosas y pueden reducir el rendimiento del sistema de manera significativa. Afortunadamente y gracias al soporte para multiprocesamiento, las máquinas modernas suelen asegurar la consistencia de cache por hardware.

Conclusiones

Las comparaciones entre DMA y PIO dependen de la arquitectura, el software y las circunstancias en las que se usan. Aunque DMA puede lograr rendimientos mayores que PIO y actualmente es el método de transferencia de la gran mayoría de las NICs para redes de alta velocidad, hay situaciones donde PIO puede ser ventajoso. Por ejemplo, PIO suele tener un overhead por byte más alto, mientras que DMA tiene más overhead por transferencia ya que la CPU y la NIC deben sincronizarse al final de una transacción, y esto generalmente se hace a través de una interrupción por hardware. Por lo tanto, PIO puede llegar a ser más eficiente para transferencias cortas y DMA lo es para transferencias largas.

2.5 La latencia de la memoria principal y la inefectividad de la cache

Al conectar la CPU con la memoria principal, el bus del sistema es el camino crítico por el cual fluyen los datos de red. Aunque pueden no pasar por el bus de Entrada/Salida, dependiendo de la arquitectura, los datos de red siempre pasarán al menos una vez por el bus del sistema para ser almacenados en memoria.

Los datos son transferidos entre la memoria cache y la memoria principal usando bloques de cache de tamaño fijo. El ancho de banda de memoria principal disponible para la CPU está limitado por la relación del tamaño de los bloques y el tiempo que demanda leer o escribir un bloque de cache. Este tiempo consiste en la latencia de acceso a memoria, más el tiempo de transferencia de un bloque de cache.

El costo y la densidad hacen que DRAM (Dynamic Random Access Memory) sea la tecnología elegida para la memoria principal en las computadoras modernas. La performance de DRAM se caracteriza principalmente por su latencia de acceso, es decir, el tiempo que le lleva colocar el dato en el bus, una vez que se le presenta una dirección al chip. Mientras que la densidad de la DRAM se ha incrementado dramáticamente en los últimos años, la latencia de acceso se ha mejorado sólo a un 7 % anual [18] y no se esperan mejoras importantes en el futuro cercano. Existen técnicas para incrementar la velocidad de transferencia (DDR, hypertransport, etc), pero ésto no resulta en un incremento importante en el ancho de banda a memoria porque la latencia de acceso continúa dominando mayormente el tiempo de transferencia para un bloque de cache fijo. Para ser efectivos, los incrementos en velocidad de transferencia de datos deberían ser combinados con un incremento en el tamaño del bloque de cache. Desafortunadamente, el tamaño del bloque de cache no puede ser modificado arbitrariamente sin afectar la tasa de aciertos del sistema de cache.

2.5.1 ¿Por que es un problema la latencia de memoria principal?

El problema fundamental es que los procesadores se están acercando a velocidades de reloj de 4 GHz, mientras que, por ejemplo, el estándar DDR2-800 (la especificación más rápida de DDR2 hasta el momento) indica una frecuencia de reloj del bus de memoria de 400 MHz, lo que representa efectivamente una relación de 10 a 1 en velocidades de reloj. Por lo tanto cuando un procesador necesita un item de dato que no está ubicado en su cache, tiene que esperar diez ciclos de reloj por cada ciclo del bus de memoria que le tome al chip de RAM el buscar y enviar los datos. Típicamente, la latencia de acceso a una palabra en RAM consume varios ciclos del bus de memoria (en particular, el estándar DDR2 es conocido por sus altas latencias de lectura, que llegan a ser de hasta 9 ciclos del bus de memoria) y luego se requiere tiempo adicional para atravesar el bus de memoria (en ciertas arquitecturas, como la del Xeon, los datos tendrán que pasar por el northbridge y ser ruteados hacia el front side bus, lo que agrega aun más latencia) hasta el procesador. Esto significa que leer datos de la RAM puede ocupar el equivalente a cientos de ciclos de reloj del procesador en los cuales no se realiza trabajo real para la aplicación.

Si el procesador incluye la tecnología Simultaneous MultiThreading (SMT), popularizada por Intel con el nombre de HyperThreading, puede dedicar sus

recursos a otro thread ejecutándose al mismo tiempo mientras se lleva a cabo el acceso a memoria principal. Sin embargo, el procesador no puede hacer nada con el paquete de red que provocó el acceso a memoria hasta que se complete la lectura de datos de memoria principal[2]. En comparación con una lectura de memoria, una computación (como decodificar el header de un paquete) ocurre a la velocidad total del procesador y por lo tanto no suelen implicar un overhead significativo.

A medida que la frecuencia del procesador crece, el tiempo de cómputo decrece pero la latencia de memoria se incrementa en términos de ciclos de CPU porque la performance de memoria se mantiene prácticamente constante.

2.5.2 Accesos a memoria principal de TCP/IP

En general, una aplicación que procesa datos de E/S causa que la CPU lea de (y posiblemente almacene en) memoria principal cada palabra de una unidad de datos, potencialmente múltiples veces. El procesamiento de los datos por la CPU (como, por ejemplo, checksums y copias físicas en memoria) incrementa el número de veces que los datos pasan a través del bus del sistema, lo cual reduce severamente el ancho de banda a memoria[3].

El camino de datos de toda transferencia por red involucra al menos tres accesos a memoria principal. Durante la recepción, la NIC hace una escritura DMA del paquete en los buffers alocados en memoria principal, luego la CPU lee los datos y los escribe en los buffers de la aplicación. Durante la transmisión la secuencia es la inversa. En el peor de los casos y dependiendo del tamaño de los paquetes con respecto al tamaño de la memoria cache, se llegan a efectuar hasta 5 accesos a memoria principal. Pueden estudiarse los detalles de los patrones de acceso a memoria de TCP/IP en [10]. En la misma publicación se plantea que el ancho de banda de memoria principal en las computadoras modernas de propósito general está en el mismo orden que el ancho de banda de las redes de alta velocidad emergentes. Incluso se estima que esta situación persistirá en el futuro cercano, ya que cuestiones de costo previenen, en general, el uso de tecnologías de memoria e interconexiones internas mucho más rápidas. Por lo tanto es necesario minimizar el número de viajes que los datos de red hacen entre la CPU y la memoria para evitar el cuello de botella.

[2]Como se verá más adelante, esto es incluso peor en el kernel Linux ya que hay sólo un thread de transmisión y uno de recepción por procesador o core, para el procesamiento de todos los protocolos y conexiones de red. Cuando un procesador accede a memoria principal por un dato de red, se paraliza completamente uno de estos threads.

[3]El ancho de banda efectivo a memoria para el procesamiento de los datos de red se verá reducido a B/N, donde B es el ancho de banda a memoria y N es el número de veces que los datos pasan por el bus.

2.5.3 Inefectividad de la memoria cache

Un sistema de memorias cache incrementa la performance del sistema de cómputo reduciendo la latencia promedio de acceso a las instrucciones y a los datos. También reduce la contención de los procesadores por el acceso a memoria principal, entre ellos mismos y con los dispositivos basados en DMA. La efectividad de la memoria cache depende de su tamaño y la organización y de la localidad en los accesos de la CPU a la memoria principal.

Pero, como se mencionó, existen factores que limitan la efectividad de la memoria cache para eliminar tráfico a memoria principal asociado con accesos a datos de red. Existen trabajos específicos que estudian el comportamiento de cache en el procesamiento de protocolos de red[20]. A continuación se consideran algunos factores que influyen en este comportamiento:

Planificación del procesador y polución de cache

El scheduling de la CPU puede causar la intercalación entre el procesamiento de una unidad de datos de red y la ejecución de otros threads. Cuando se retoma el procesamiento original, las porciones de cache que contenían la unidad de datos pueden haber sido reemplazadas. Esto se conoce con el término de polución de memoria cache. Como se estudiará más adelante, la gran cantidad de interrupciones por hardware relacionadas con la red y los eventos que éstas señalizan disparan threads que causan el rescheduling del procesador, incrementando los problemas de polución.

Además, el software de red utiliza colas para comunicar distintas etapas de procesamiento de un paquete. Típicamente, el encolado ocurre en el límite usuario/kernel, entre ciertos protocolos, y entre el top half y el bottom half del manejador de interrupciones (explicado en el próximo capitulo). En el capítulo 5 se explica por qué el encolado y la sincronización en sistemas SMP perjudican el comportamiento de cache.

Tamaño de la Cache

Las memorias cache, particularmente las caches on-chip, están limitadas en tamaño. En la práctica, su tamaño efectivo se reduce aun más debido a la asociatividad limitada en las organizaciones de mapeo directo y asociatividad por conjuntos. Para que los datos permanezcan válidos en cache durante una operación que involucra cargar cada palabra en un registro y almacenarla en una ubicación diferente de la memoria principal, la cache debe ser al menos dos veces el tamaño de la unidad de datos. En la práctica, los requerimientos de tamaño de la cache son más exigentes, debido a los accesos a otras variables durante la manipulación de los datos y el acceso a datos globales de una conexión. Sin

embargo, las unidades de datos de red tienden a ser de mayor tamaño por cuestiones de eficiencia (se reducen los costos por paquete, estudiados en el capítulo 4).

Organización de la Cache

Las caches que son virtualmente indexadas no requieren una traducción de direcciones virtuales a físicas para acceder a los datos cacheados. Con esta aproximación los datos cacheados pertenecientes a páginas virtuales compartidas no pueden permanecer válidos entre los limites de dominios de protección (ver capítulo 4). En otras palabras, los datos deben ser releidos luego de un cambio de contexto a un dominio de protección diferente, incluso aunque los buffers de datos fueran físicamente compartidos entre los dominios. Las caches físicamente indexadas no presentan este problema.

Política de escritura de cache

Las caches de datos en los monoprocesadores usualmente utilizaban una política write-though, lo que significa que cada operación de almacenamiento requiere una escritura en memoria principal. Por lo tanto, muchas escrituras consecutivas, como ocurre cuando se lee y escribe cada palabra de una unidad de datos en un buffer de memoria principal, hace que el procesador pierda ciclos útiles accediendo a memoria.

Las arquitecturas SMP utilizan otras reglas que aseguran la coherencia entre las caches y la memoria principal, aunque también presentan algunos problemas como se verá más adelante.

2.5.4 La regla 1GHz/1Gbps

Existe una regla generalmente aceptada que indica que para alcanzar 1 Gbps de capacidad de transmisión de un enlace se requiere aproximadamente 1 GHz de velocidad de procesamiento de CPU. Esta regla se ha validado a lo largo de los años, aunque sólo para transferencias con paquetes de tamaños grandes. Para transferencias con unidades más pequeñas, se ha encontrado que los requerimientos de procesamiento pueden ser 6 o 7 veces mayores.

Incluso peor, existen trabajos[10] que demuestran que el procesamiento de red no escala proporcionalmente con la velocidad de la CPU. De hecho, el consumo de CPU por byte se incrementa cuando la frecuencia de la CPU es mayor. Esto ocurre porque mientras crece la velocidad de la CPU, la disparidad entre la latencia de la memoria y la I/O por un lado, y la velocidad de la CPU por el otro se intensifica cada vez más.

Como se estudió previamente, el procesador debe esperar sin realizar trabajo mientras accede a la memoria o a los dispositivos de I/O. El tiempo ocioso a esa

granularidad tan fina no es detectado por el sistema operativo y, por lo tanto, no se invoca al scheduler. Esos ciclos desperdiciados son contabilizados como si fuera trabajo realizado y se suman al tiempo de utilización de la CPU. Tomando medidas a dos frecuencias de CPU (800 MHz y 2.4 GHz) se ha mostrado que la performance de red sólo está escalando en un 60 % con respecto al incremento de la frecuencia. En conclusión, a medida que la velocidad de la CPU se incrementa la regla 1 GHz/Gbps no se mantiene, sino que empeora.

En general, la tecnología SMT ayuda a disminuir los ciclos de CPU ociosos en los accesos a memoria de las aplicaciones con más de un thread de ejecución. Sin embargo, debido a la reducida localidad de las referencias a memoria principal de las implementaciones de los protocolos y a la granularidad gruesa con la que se paralelizan los protocolos en Linux (y la mayoría de los sistemas operativos), la poca performance de memoria principal sigue siendo unos de los problemas estructurales más importantes en las arquitecturas modernas de las computadoras conectadas a la red.

Capítulo 3

El Subsistema de red en el kernel Linux 2.6

3.1 Introducción

En este capítulo se discute el procesamiento tradicional que se realiza en una pila de protocolos de red basada en la semántica UNIX y dirigida por interrupciones de hardware, como es la pila TCP/IP en el kernel Linux. La arquitectura de procesamiento de red de Linux se construye como una serie de capas de software conectadas de manera similar a la arquitectura en capas de la pila de protocolos.

Como se observa en otras partes del código del kernel, el subsistema de red utiliza un diseño orientado a objetos (con las limitaciones propias del lenguaje C), donde las clases base se especializan para soportar a cada familia de protocolos pero utilizan siempre los mismos métodos (el polimorfismo se implementa con punteros a funciones). La capa de sockets BSD es la interfaz genérica de sockets tanto para la comunicación por red como para la comunicación entre procesos. En el caso de TCP/IP se deben invocar a las rutinas de la capa de sockets INET para efectuar la comunicación usando los protocolos de Internet.

Por otro lado y como sucede en otros subsistemas del kernel Linux, los detalles de implementación de los drivers de dispositivos están ocultos del resto de las capas de red. La pila TCP/IP provee un mecanismo de registración entre los drivers y las capas superiores, de manera que es posible invocar a la función de transmisión del driver para una interfaz de red dada sin conocer sus detalles internos.

Como se mencionó anteriormente, el objetivo fundamental de este trabajo se centra en los mecanismos del sistema operativo que dan soporte al procesamiento de red, mientras que el estudio detallado de la implementación de los protocolos

involucrados está fuera del alcance de esta tesis. Por lo tanto, antes de describir los pasos en el procesamiento de red, es necesario mencionar algunos de estos mecanismos que, como se verá más adelante, influyen de manera decisiva en la eficiencia con que se implementan los protocolos.

3.2 Mecanismos de Linux involucrados en el procesamiento de red

A continuación se estudia la forma en que 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. También se presentan algunas estructuras de datos que influyen significativamente en la forma de procesar el tráfico de red en Linux.

3.2.1 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 pueda 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 ahí 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 [1].

[1] Algunas implementaciones primitivas de TCP/IP o diseñadas para entornos embebidos completan el procesamiento de un paquete entrante en el contexto del manejador de interrupción. Sin embargo los sistemas operativos modernos suelen utilizar estas capacidades de threading.

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[2] 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, realizar el procesamiento de capas superiores, 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.

Adicionalmente, 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 preferible 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[3]. 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

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

[3]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[7].

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 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 deben 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 los drivers de las NICs, 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 [25].

Los threads del kernel y ksoftirqd

Los sistemas Unix tradicionales delegan algunas tareas críticas, como escribir del buffer-cache al disco o hacer el swapping de páginas poco referenciadas, a procesos que se ejecutan intermitentemente en background. Los sistemas operativos modernos utilizan en su lugar threads del kernel, los cuales no necesitan manejar un contexto en modo usuario. En Linux, los threads del kernel se ejecutan sólo en modo kernel, mientras que los procesos normales se ejecutan alternativamente en modo kernel y modo usuario. Sin embargo, sigue siendo el scheduler el encargado de decidir el momento en el que se ejecutarán estos threads y el momento en el que dejarán de ejecutarse para darle lugar a otra tarea.

En particular, existe un grupo de threads del kernel (uno asociado a cada procesador de la máquina, y se los denomina ksoftirqd_CPU0, ksoftirqd_CPU1, etc) a los cuales se les asigna la tarea de verificar si existen softirqs que hayan quedado sin ejecutarse, en cuyo caso se invocan los manejadores apropiados durante el tiempo permitido hasta que tengan que devolver el control de la CPU a otras actividades [4] (por ejemplo, si el scheduler ejecutado en la interrupción del timer marca al thread actual para indicar que ha usado su cuota de tiempo de ejecución) o hasta que no haya más softirqs pendientes para manejar.

Bajo cargas intensas de red, básicamente se termina delegando al scheduler la función de intercalar procesamiento de red con el resto de los procesos. Ésta interferencia no sólo no es deseable con respecto al usuario, sino que parece ser una solución no óptima desde el punto de vista del scheduling. Cuando las actividades de red estén entre las operaciones más comunes de una máquina, como se espera que en un futuro cercano, siempre habrá procesamiento de red pendiente y cada posible invocación a un manejador de softirq se hará efectiva, interrumpiendo a la CPU. Pero, incluso peor es el hecho de que el mismo sche-

[4] La prioridad de un proceso, también conocida como prioridad nice, es un número que va desde -20 (máximo) hasta 19 (mínimo). Los threads ksoftirqd poseen prioridad 19. Esto es así para evitar que softirqs que se ejecutan frecuentemente, como NET_RX_SOFTIRQ, no puedan ocupar completamente las CPUs, lo que dejaría sin recursos a los otros procesos.

duler será el encargado de tomar decisiones de procesamiento de red, que poco tiene que ver con los patrones de cómputo de los proceso de usuario que éste planifica, sin hacer ningún tipo de distinción entre uno y otro.

Como se propone en este trabajo, un kernel asimétrico puede aislar completamente el procesamiento de red del resto de los procesos. Pero además, resulta interesante plantear la posibilidad de un scheduler específico que realice el trabajo de planificar threads que atiendan conexiones de red, en función de los recursos disponibles, las prioridades, las políticas de QoS, etc., sin interferir en el funcionamiento del resto de los componentes del sistema.

3.2.2 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. La softirq de recepción accede para obtener las tramas recibidas por la NIC y colocadas en esta estructura por el manejador de interrupciones. La softirq de transmisión la 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. 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).

En la versión 2.5, fue introducido un nuevo mecanismo para el manejo de tramas entrantes en el kernel Linux, conocido con el nombre de NAPI (por Nueva API). 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 coloca el descriptor de la NIC en la lista apuntada por poll_list de la estructura softnet_data correspondiente. En las siguientes invocaciones a net_rx_action, cada NIC en la lista poll_list es verificada hasta que no haya más tramas que recibidas, entonces se reactivan las interrupciones. Este método híbrido logra reducir el uso de CPU en el procesamiento de recepción.

3.2.3 Los socket buffers

Los socket buffers (sk_buff) de Linux 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.

3.3 El procesamiento de red

El procesamiento TCP/IP puede ser clasificado en tres categorías principales: procesamiento de la conexión, procesamiento de recepción y procesamiento de transmisión.

3.3.1 Procesamiento de conexión

El procesamiento de la conexión se refiere al establecimiento y terminación de la conexión TCP entre los sistemas finales que se comunican. Una vez que el sistema establece una conexión, la transferencia de datos ocurre sobre la conexión a través de operaciones de transmisión y recepción llamadas colectivamente como *data-path processing* o *fast path*.

Suponiendo que una conexión ha sido establecida entre dos nodos, a continuación se describe el procesamiento involucrado en los caminos de comunicación de recepción y envío.

3.3.2 Procesamiento de recepción

El procesamiento de recepción comienza cuando la NIC recibe una trama Ethernet de la red. En la inicialización, el driver provee de descriptores a la NIC, los cuales están típicamente organizados en anillos circulares en memoria RAM, y a través de estos descriptores el driver le informa a la NIC la dirección de memoria del buffer donde se debe almacenar los datos del paquete entrante. Para extraer el paquete embebido en la trama, la NIC remueve los bits Ethernet y actualiza el descriptor con la información del paquete.

La NIC efectúa una operación DMA para copiar los datos entrantes en este buffer y, una vez que coloca el paquete en memoria, actualiza un campo de estado del descriptor para indicarle al driver que ese descriptor tiene un paquete válido. Después genera una interrupción para iniciar el procesamiento del paquete recibido a través del manejador de interrupciones de la NIC. Éste debe alocar un socket buffer y colocar la trama recibida dentro de esa estructura (o en lugar de alocar buffers al recibir cada paquete e incrementar el tiempo en interrupción, los buffers pueden ser pre-alocados y continuamente monitoreados para asegurarse de que no sean consumidos en su totalidad). Los socket buffers recibidos luego son encolados en la estructura sofnet_data y se planifica la softirq de recepción en la CPU que recibió la interrupción. Cuando el manejador net_rx_action() es elegido para ejecutarse, realiza el resto del procesamiento de los protocolos IP y TCP o UDP, hasta colocar los datos en la cola del socket.

Ya en el contexto de la softirq, el próximo paso es identificar la conexión a la cual pertenece este paquete. TCP/IP guarda la información de estado de cada conexión en la estructura de datos denominada TCP/IP Control Block o TCB. Como puede haber muchas conexiones abiertas y por lo tanto muchos TCBs en memoria, la pila usa un mecanismo de hashing para realizar una búsqueda rápida del TCB correcto, calculando el valor de hash a partir de la dirección IP y los números de puerto origen y destino de la conexión. Entonces, se actualizan varios campos en el TCB, como el número de secuencia de los bytes recibidos y confirmados.

Si la aplicación ya ha asignado un buffer para recibir los datos entrantes, la pila TCP/IP copia los datos directamente desde el buffer de la NIC al buffer de la aplicación. En otro caso, se almacenan los datos en un buffer temporal para una entrega posterior a la aplicación. Cuando la aplicación hace una llamada a la system call recv() (o read()), estos datos son copiados a los buffers de la aplicación. Realizar una copia de memoria desde un buffer del kernel al buffer de la aplicación es uno de las operaciones más costosas en tiempo en el procesamiento de recepción (como se verá en el capítulo 8).

3.3.3 Procesamiento de transmisión

El procesamiento de transmisión comienza cuando una aplicación envía un buffer de datos a la pila TCP/IP, a través de las system calls send() o write(). La aplicación pasa como parámetro un id de socket junto con el buffer de datos, y la pila TCP/IP usa este id de socket para localizar el TCB de la conexión. La pila TCP/IP luego puede copiar los datos de la aplicación en un buffer interno para prevenir que los datos sean sobreescritos por la aplicación antes que sean enviados. La system call de envío retorna en este punto. Mientras el sk_buff atraviesa las capas de protocolos, se va agregando la información correspondiente.

Cuando el tamaño de ventana del receptor indica que es tiempo de transmitir datos, la pila TCP/IP divide los datos acumulados en segmentos MTU (Maximum Transfer Unit). Un MTU es típicamente de 1460 bytes en LANs Ethernet. Luego se computa el header para cada segmento, 20 bytes para TCP y 20 bytes para IPv4, y se concatenan los segmentos con los headers.

Primero se verifica si el socket buffer está dividido en un conjunto de fragmentos[5] y si la interfaz soporta E/S tipo Scatter/Gather. Si lo primero es verdadero y lo último es falso, es necesario linealizar el paquete. Luego se analiza si el paquete necesita checksum y si la interfaz puede calcular el checksum por hardware y, en caso de ser necesario, se realiza el cálculo por software.

Eventualmente, dependiendo de la disciplina de colas utilizada en ese momento, el estado de la cola y del scheduler de tráfico activo, se invoca a la función de transmisión del driver correspondiente. Esta función normalmente coloca el socket buffer en una cola privada al driver y emite el comando de transmisión a la NIC, el cual se ejecutará asincrónicamente, efectuando una operación DMA desde el buffer en memoria RAM hasta la memoria FIFO en la NIC, y será sucedido por una interrupción de hardware para que, finalmente, el driver pueda liberar recursos y realizar las tareas de mantenimiento necesarias (generalmente esto se hace en la softirq net_tx_action()).

3.4 Identificando componentes del procesamiento TCP/IP

Como se muestra en la figura 4.1 (adaptada de [22]), se pueden identificar cinco componentes distintos en el procesamiento tradicional de TCP/IP mencionado anteriormente. Es interesante tener presente esta división funcional por varios motivos. Por un lado, nos permite identificar con precisión la fuente del overhead generado por las operaciones de red. Pero aún más importante para este trabajo, es tener la posibilidad de plantear diferentes alternativas de distribución del procesamiento de los componentes entre CPUs. En el capítulo 8, donde se explica

[5] un socket buffer segmentado consiste en un único encabezamiento IP y una lista de fragmentos que los usarán

el prototipo implementado, se verá que algunos componentes del procesamiento TCP/IP se harán de manera exclusiva en un procesador dedicado a tal fin.

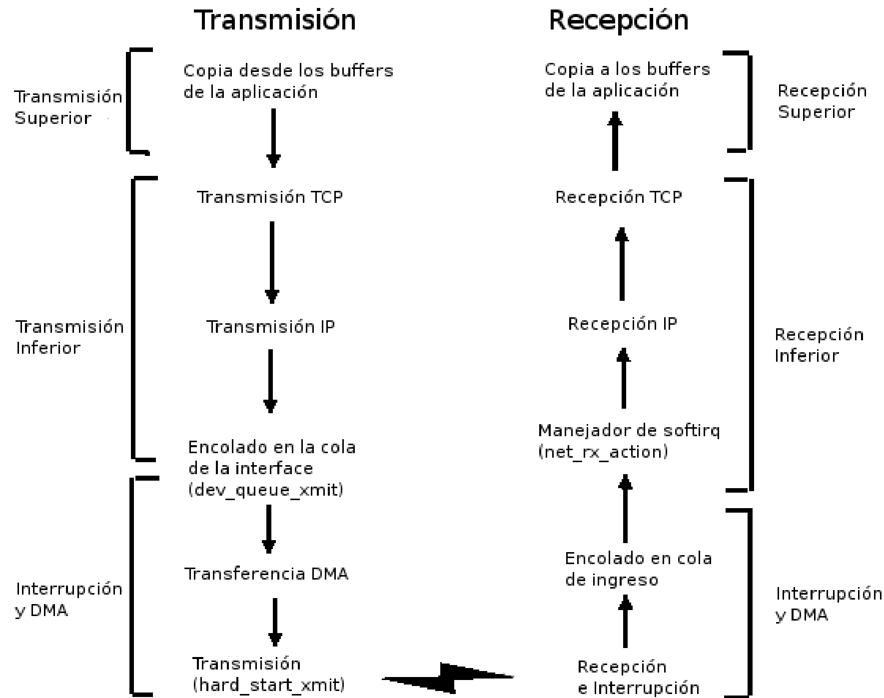

Figura 3.1: Componentes del procesamiento TCP/IP

- Procesamiento de interrupciones: El componente de procesamiento de la interrupción incluye el tiempo que toma manejar las interrupciones de la NIC e invocar 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 interfaz 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 estructura 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: Aquí 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 retransmisiones. 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 interfaz.

Capítulo 4

Clasificando los costos del procesamiento de red

4.1 Introducción

En términos generales, existen dos tipos de overhead en el procesamiento de entrada/salida de red. Uno es el *data-touching overhead* (DTO) o costo por byte y el otro es *non data-touching overhead* (NDTO) o costo por paquete. A continuación, denominaré operaciones DTO a aquellas que generan costo por byte y operaciones NDTO a las que generan costo por paquete.

Las operaciones DTO, como el checksum y copias por software, contribuyen a aumentar el tráfico de memoria porque cada byte de datos necesita recorrer el bus de memoria al menos una vez. Por ejemplo, el checksum requiere que cada byte sea leído de la memoria y la copia requiere que cada byte sea leído de una fuente en memoria y escrito en un destino en memoria. Las operaciones NDTO principalmente involucran procesamiento de CPU, lo cual incluye procesamiento de protocolos, cambios de contexto, manejo de interrupciones, manejo de buffers, etc. A continuación, se mencionan estos tipos de overheads profundizando en aquellos generados por mecanismos propios del sistema operativo.

4.2 Operaciones DTO

4.2.1 Transferencia de datos entre dominios de protección

Los mecanismos de protección de recursos en los sistemas operativos requieren transferencias de datos entre diferentes dominios de protección. En el caso más simple, un flujo de datos de E/S es usado por una única aplicación corriendo

sobre el kernel de un sistema operativo monolítico tradicional. En general, ciertos procesos de usuario adicionales, como manejadores de ventanas, servidores de multimedia, o los procesos *servers* de un sistema operativo basado en una arquitectura de microkernel (estudiado en el capítulo 6) pueden introducir más límites de protección que los datos de E/S deberán cruzar en su camino.

La copia de datos por software, como una forma de transferir datos a través de los límites de protección, profundiza el problema del cuello de botella de memoria, por lo tanto es esencial realizar esta operación de manera eficiente. El diseño de una operación de este tipo puede ser analizado basándose en los siguientes factores: su semántica, el modelo de transferencia, el método de transferencia y las estructuras de datos utilizadas.

Existen tres modelos para transferir datos entre dominios:

- En el modelo de copia, los datos son copiados de un dominio al otro, es decir, la copia original se preserva en el dominio de origen luego de la transferencia. Es semánticamente simple y flexible, aunque difícilmente se pueda implementar eficientemente.

- En el modelo de movimiento, los datos son removidos del origen y colocados en el dominio destino. En este modelo, el dominio origen pierde el contenido del buffer luego de la operación (deberá hacer copias privadas si las necesita).

- En el modelo compartido, luego de la transferencia, ambos dominios tienen acceso a los mismos datos. Cualquier modificación hecha a los datos por uno de los dominios será visible por el otro.

Por otro lado, existen dos métodos para la transferencia de datos entre dominios:

- El método de transferencia física involucra un movimiento de datos en memoria física (mover cada palabra de los datos desde la memoria física del dominio origen hacia la memoria física del dominio destino). Éste es un método flexible ya que la granularidad de la transferencia es de un byte (o una palabra) y, por lo tanto, se pueden transferir datos desde cualquier ubicación y de cualquier tamaño hacia cualquier lugar en la memoria. Sin embargo presenta una desventaja fundamental: un gran overhead en tiempo y espacio. La transferencia física de una palabra implica dos accesos a memoria.

- El método de transferencia virtual involucra el movimiento de los datos en memoria virtual. En otras palabras, la transferencia mapea una región en el dominio de destino con las páginas físicas que contienen los datos a ser transferidos y que ya están mapeadas en la memoria virtual del dominio

origen. A estas páginas se las conoce como páginas de transferencia. En transferencias virtuales, la granularidad de la transferencia es una página física. Los datos transferidos deben estar contenidos en una o más páginas de transferencia exclusivamente y estas páginas no deben contener otros datos.

Por razones de privacidad, el kernel debe limpiar (llenar con ceros) la porción no usada de una página recién alocada como página de transferencia y ésto puede ser sustancialmente costoso. Ya que la unidad de transferencia es una página completa, la dirección de los datos de destino debe estar en el mismo offset relativo al límite de la página. Además el tamaño del espacio de destino debe ser el tamaño del número de páginas transferidas, el cual normalmente es mayor que el tamaño de los datos. El uso parcial de las páginas de memoria requerirá más páginas por dato de red, resultando en un mayor consumo de memoria física, mayor overhead de alocación de páginas (y su remapeo) y mayor demanda de líneas de TLB.

La transferencia física normalmente es aplicada al modelo de copia. Una alternativa es la copia virtual o movimiento virtual. La copia virtual, según el modelo, mapea una región en el dominio de destino con las páginas de transferencia, mientras que no afecta al mapeo con estas páginas en el origen. Si los procesos en ambos dominios sólo leen los datos, la copia virtual es una solución aceptable. Si alguno de los dominios intentara modificar los datos, sería necesario realizar una copia física de la página que contiene los datos para que las modificaciones no afecten los datos del otro dominio. Esta técnica es conocida como *Copy-On-Write* o COW y se usa actualmente en el kernel Linux, por ejemplo, para diferir en el tiempo la copia del espacio de direccionamiento de un proceso padre al proceso hijo cuando se invoca a la llamada *fork()*.

El movimiento virtual desvincula las páginas de transferencia de la memoria virtual del dominio de origen y las mapea en el dominio de destino. A diferencia de la copia virtual, no es necesario implementar COW, con lo cual este mecanismo es relativamente sencillo.

Tanto el movimiento virtual con el remapeo de páginas, como la copia virtual con COW requieren una implementación cuidadosa para lograr una latencia aceptable. Ya que prácticamente todo sistema operativo moderno emplea un sistema de memoria virtual con dos niveles (algunos con tres; en Linux ésto depende de la arquitectura), los remapeos requieren la modificación tanto de las tablas de páginas de bajo nivel y dependientes de la máquina, como de las estructuras de alto nivel independientes de la máquina (llamadas directorios en Linux)[1], además de las actualizaciones necesarias en las tablas de la TLB.

[1] Existe una técnica denominada *Lazy Page Remapping* que actualiza las estructuras de alto nivel durante el remapeo y sólo modifica las tablas de página de bajo nivel en el manejador de fallos de página.

El tiempo que le lleva cambiar a modo kernel, tomar los locks necesarios para manipular estructuras de datos de la memoria virtual, modificar los mapeos (tal vez a diferentes niveles) para cada página, realizar las acciones de consistencia necesarias con la TLB y con la cache, y volver al modo usuario representa una limitación importante a estas soluciones.

En el modelo compartido, la memoria virtual compartida hace que se asocien regiones en el origen y en el destino a las mismas páginas de transferencia y de manera estática. Como todos los dominios tienen acceso de lectura y escritura a estas páginas existen varios problemas: principalmente aquellos relacionados con la seguridad y el hecho de que no siempre es posible evitar las copias físicas (por ejemplo si los datos no son consumidos inmediatamente o si fuera necesario transferirlos hacia otro dominio, entonces deben ser copiados físicamente desde la región compartida a otro destino).

Por otro lado la transferencia debe organizarse de alguna manera conocida por los distintos dominios, y esta organización también puede requerir copias físicas afectando a la performance. Por ejemplo, si se utiliza la región compartida como un buffer no estructurado y los datos a transmitir se representan en una estructura auto-referenciada (como un árbol o una lista enlazada), puede ser necesario copiar físicamente los datos para linealizar la información en un arreglo contiguo en memoria física.

Sin embargo, también es posible utilizar estructuras de datos más complejas para implementar la región compartida. En estos casos los datos transferidos no son almacenados en un arreglo simple con lo cual el dominio de destino debe conocer los métodos para acceder a los datos, de acuerdo con las estructuras utilizadas (esto puede transmitirse como parte de los datos). Además, si los datos contienen punteros a direcciones virtuales, es necesario una traducción al nuevo espacio de direcciones del destino.

Una solución intermedia es la que se conoce como datos semiestructurados, que consiste en punteros a arreglos no estructurados de datos. Por un lado no requiere la operación previa de linealización y, aunque tiene punteros asociados, éstos no están embebidos en los datos por lo que pueden ser localizados rápidamente para una posible traducción. Además el método de acceso suele ser conocido con lo que no es necesario transmitirlo como parte de la transferencia. Por último los datos semiestructurados suelen ser altamente compatibles con las operaciones scatter-gather para transferencia DMA con el dispositivo de red. Linux ofrece un conjunto de llamadas al sistema que utilizan esta técnica para el intercambio de datos entre el espacio de usuario y el kernel.

4.2.2 Application Programming Interfaces

La API que el kernel ofrece al espacio de usuario define la base de los servicios y operaciones que el sistema operativo provee a los procesos. En particular, define

la sintaxis y semántica de las operaciones exportadas por el kernel, llamadas normalmente *system calls*. Los buffers para entrada/salida de datos aparecen en esta definición como argumentos a las operaciones. La semántica de pasaje de parámetros definida por la interfaz puede tener un impacto significativo en la eficiencia de la transferencia de datos entre el kernel y la aplicación.

Considérese, por ejemplo, las system calls de UNIX read() y write(). Estas operaciones especifican un argumento puntero que referencia a la dirección de un buffer contiguo en el espacio de direccionamiento de la aplicación, y un argumento entero que representa el tamaño del buffer. Las aplicaciones pueden elegir cualquier dirección en su espacio de direccionamiento, sin restricciones en el tamaño y la alineación del buffer. La semántica de estas operaciones especifica que los datos son copiados desde y hacia el buffer de la aplicación durante la operación. Esto es, durante una operación read() el buffer de la aplicación es sobreescrito con los datos de entrada, y luego de que se completa una operación write() la aplicación es libre de reusar el buffer.

La representación de bajo nivel de los buffers de datos y la semántica de las operaciones read y write hacen que sea difícil lograr una implementación que evite la copia física de los datos por los siguientes motivos.

En primer lugar, todas las técnicas basadas en memoria virtual para transferencias entre dominios operan a una granularidad de página. Si la primera y última dirección del buffer del usuario no están alineadas con los límites de páginas, el sistema debe tomar recaudos con respecto a las porciones de la primera y última página de transferencia que no están ocupadas por el buffer. Esto se explicó previamente en este capítulo.

Por otro lado, la semántica de la operación write() permite que el proceso de usuario modifique el buffer inmediatamente después de que la llamada retorna. Por lo tanto el sistema debe o bien copiar la pagina afectada o bloquear al proceso hasta que el sistema operativo concluya la operación con los datos de salida. La última aproximación puede degradar la tasa de transferencia efectiva de E/S, aun cuando evite la copia, porque impide el solapamiento de E/S con procesamiento de la aplicación.

Finalmente, las system calls read() y write() especifican un único buffer de datos contiguo. Los datos que arriban a un dispositivo de red suelen ser distribuidos en memoria principal por cuestiones de performance (por la función de scatter-read, mencionada en el capítulo 2) o debido a la fragmentación de paquetes de red. Si una operación read() especifica una cantidad de datos que involucra a varios fragmentos en memoria del kernel, la copia física para linealizar el buffer de datos en la memoria de la aplicación es inevitable.

En resumen, tres problemas se asocian con las system calls tradicionales de Unix, read() y write(), relacionados con evitar las copias de datos en memoria principal. Estas system calls permiten buffers de datos con una alineación y tamaños arbitrarios, requieren buffers de datos contiguos y tienen una semántica

de copia. Afortunadamente, ya se están planteando APIs alternativas orientadas a incrementar la eficiencia en su operación. En el capítulo 7 se presentan algunas de estas ideas, relacionadas principalmente con la posibilidad de realizar operaciones asincrónicas (no bloqueantes), vectorizadas (utilizando más de un buffer de datos por operación) y utilizando memoria compartida entre el kernel y el espacio de usuario.

4.2.3 Manipulación de datos

Son computaciones en las cuales se inspecciona y posiblemente modifica cada palabra en un paquete de red. Si consideramos sólo el rendimiento *end-to-end*, la manipulación de datos principalmente involucra al checksum. Sin embargo, se suelen realizar manipulaciones más complicadas y costosas como conversiones en la capa de presentación, encriptación, compresión, detección y corrección de errores, etc. La manipulación de datos puede realizarse por hardware o software. El soporte por hardware puede reducir la carga de la CPU y, cuando se integra apropiadamente, logra reducir el tráfico de memoria. Un ejemplo es el checksum soportado en hardware. Sin embargo, como se ha comprobado recientemente (y se explica más adelante), los beneficios de esta solución no son tan importantes como se pensaba.

Las manipulaciones de datos de red por software típicamente acceden a los datos de manera independientemente una de otra ya que generalmente son parte de distintos módulos de software que incluso pueden residir en dominios de protección separados (kernel, servers, aplicaciones). Las referencias a memoria resultantes tienen poca localidad y, por lo tanto, requieren que los datos sean leídos y escritos en memoria principal desaprovechando los beneficios de la memoria cache. Es por esto que se han intentado nuevas formas de implementar estas manipulaciones de datos de red, como se estudia en el capítulo 7.

4.2.4 Transferencia de los datos desde la NIC a la memoria principal utilizando la técnica PIO

Como se estudió en el capitulo 2, esta técnica implica la participación de la CPU para el movimiento de cada palabra de datos entre un origen y un destino. El hardware moderno suele utilizar DMA como técnica principal, aunque se continúan evaluando las ventajas y desventajas de estos métodos en diferentes situaciones y arquitecturas.

4.3 Operaciones NDTO

4.3.1 Creación y manipulación de buffers de red

Se incluye en esta categoría a la manipulación de varias estructuras de datos necesarias para la operación del subsistema de red. Principalmente, se pueden mencionar la alocación y edición de diferentes tipos de buffers (por ejemplo, los socket buffers mencionados en el capítulo 3) y colas (como las colas de defragmentación IP y colas de las interfaces de red).

Los protocolos de red como TCP/IP reciben, transmiten y procesan datos en paquetes independientes de un tamaño relativamente pequeño. En el peor de los casos, los sistemas operativos deben alocar buffers de memoria para los paquetes a medida que estos son recibidos, y deben liberar los buffers a medida que los paquetes son transmitidos, con una frecuencia cada vez mayor, según se incrementan las velocidades de transmisión. La alocación de memoria es un factor fundamental en la performance de toda pila TCP/IP, por lo tanto, el esquema de alocación de memoria debe estar basado en un método flexible y de baja latencia. La mayoría de las implementaciones TCP/IP tienen un mecanismo de alocación de memoria que es independiente del sistema operativo, sin embargo, los desarrolladores de Linux adoptaron una aproximación diferente.

La forma más simple de alocación se podría implementar a partir de un esquema de *heap*, pero esto tiende a generar fragmentación externa. Cuando los buffers son rápidamente alocados y liberados, el espacio de memoria puede verse fragmentado en pequeñas secciones, con lo que el tiempo requerido para buscar un buffer disponible empeorará. Algunas implementaciones TCP/IP solucionan el problema de la fragmentación usando un sistema separado de buffers de tamaños fijos para la pila TCP/IP, como algunas variantes de BSD.

Linux utiliza un método de alocación de memoria llamado *Slab Allocation*. Este método fue introducido en el sistema operativo Solaris 2.4 y se utiliza para alocar memoria en todos los subsistemas del kernel, y no únicamente para buffers de red. El sistema slab se organiza en caches específicas llamadas *slab caches* y hay una por cada estructura importante en el kernel. Cuando se utiliza para la alocación de buffers de red, la slab cache tiene ventajas sobre los métodos alternativos[7, 13]. Es menos tendiente a la fragmentación, es más eficiente en tiempo y hace mejor uso de la memoria cache. Además, cuando un buffer es desalocado se devuelve a la slab cache de la cual se obtuvo entonces, como cada cache es específica para un propósito, los campos comunes de las estructuras a alocar pueden ser pre-inicializados. Esto incrementa la eficiencia notablemente, ya que los sockets buffers son estructuras complejas que contienen spinlocks, contadores de referencias, y otras estructuras embebidas que requieren inicialización.

Por otro lado, ya que las operaciones de edición de buffers ocurren frecuen-

temente en la pila TCP/IP, éstas son un componente importante en el estudio de la performance. La edición de buffers (que se diferencia de la manipulación de datos) puede ser expresada como la combinación de operaciones para crear, compartir, separar, concatenar y destruir buffers. Cuando se implementan ingenuamente, estas operaciones pueden requerir copias físicas. Afortunadamente, en los sistemas operativos modernos se suelen usar (siempre que sea posible) punteros para implementar las operaciones de edición, por lo tanto el manejo de buffers se considera como parte del NDTO.

4.3.2 Operación del sistema operativo

Estos costos son los que más atención reciben en este trabajo, sin embargo, su análisis se distribuye a lo largo de todos los capítulos. En esta categoría se incluye el overhead de las system calls y la capa de sockets, el costo de la sincronización, el tiempo necesario para dormir y despertar procesos, las operaciones de scheduling de softirqs, el manejo de interrupciones, etc. En general, se miden globalmente estos costos (además de otros) con una métrica conocida como intrusión del sistema operativo (estudiada en el capítulo 6), y son los motivos por los cuales se están comenzando a plantear alternativas de diseño de sistemas operativos teniendo en cuenta los requerimientos de las redes de alta velocidad.

4.3.3 Operaciones específicas de los protocolos

Son los costos generados por aquellas tareas como calcular los campos de los headers, mantener el estado del protocolo, verificar que existe una ruta para una conexión, etc. Aunque TCP es un protocolo complicado, se ha afirmado que puede soportar altas velocidades si se implementa correctamente. Se intenta demostrar que no es el algoritmo propuesto por el protocolo sino su implementación el factor limitante con respecto la eficiencia y escalabilidad. Sin embargo, la implementación de TCP requiere un soporte fundamental del sistema operativo, por lo que los mecanismos de éste están ligados directamente a la eficiencia lograda por el protocolo.

4.4 Costos DTO versus costos NDTO

Como se indica en [16], el costo de las operaciones DTO y NDTO escala diferente con respecto al tamaño de los paquetes. El costo DTO crece linealmente con el tamaño de los paquetes y el costo NDTO se mantiene constante. Por esta razón se estudia el uso de paquetes de mayor tamaño para amortizar el costo por paquete. Un ejemplo práctico de este concepto es el uso de tramas Ethernet de gran tamaño en redes de alta velocidad, como se propone con las denominadas Jumbo Frames (explicadas en el capítulo 7). A su vez, un mayor tamaño de

paquete incrementa el costo DTO, y por lo tanto la latencia, por lo que es necesario encontrar el tamaño óptimo.

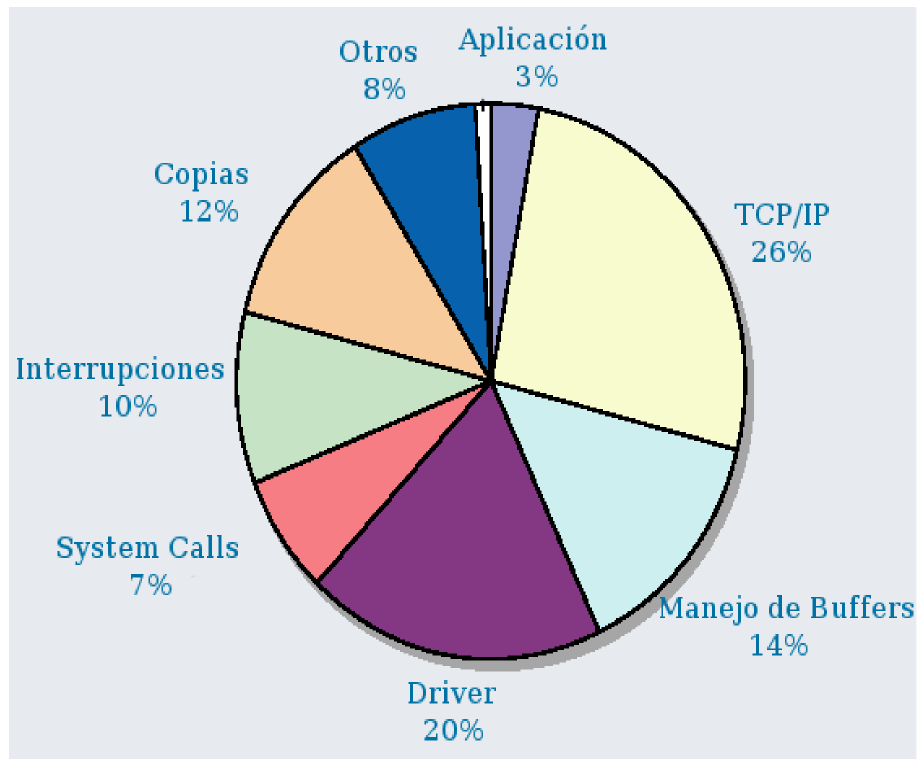

Figura 4.1: Distribución de los costos de procesamiento de TCP/IP

Aunque durante mucho tiempo los trabajos de investigación se enfocaron en reducir los costos DTO, éstos son más importantes cuando se consideran tamaños relativamente grandes de paquetes. Sin embargo, cuando se consideran los costos agregados generados en el procesamiento de paquetes con una distribución de tamaños similar a la de una situación real, el costo NDTO puede consumir una gran parte del tiempo de procesamiento.

Si se tiene en cuenta que la mayoría del tráfico TCP y UDP en una LAN tradicional se constituye de mensajes pequeños, de 200 bytes aproximadamente, y los paquetes IP que viajan a través de las WANs no suelen superar los 580 bytes[16], se vuelve imperiosa la necesidad de estudiar en profundidad las alternativas para reducir los costos NDTO. La conclusión importante para este trabajo es que se destaca la importancia de comprender y reducir el overhead generado por las operaciones y los mecanismos propios del sistema operativo, como el scheduling y sincronización de threads y procesos, el método de alocación

de memoria, la capa de systems calls, el manejo de interrupciones, etc.

Capítulo 5

La escalabilidad de TCP/IP en Linux para arquitecturas SMP

5.1 Introducción

Las arquitecturas paralelas existentes en la actualidad ofrecen una forma de resolver los altos requerimientos de procesamiento que implican el envío y recepción de datos a altas velocidades. Desde hace tiempo, SMP es el estándar en servidores de alta performance, sin embargo, últimamente también se ha comenzado a evaluar el uso de este tipo de arquitecturas en dispositivos de red dedicados, como routers y switches.

Luego de años de desarrollo orientado al enrutamiento de tráfico basado en hardware, los proveedores de routers han adoptado plataformas de propósito general como la base para sus sistemas. Se están utilizando procesadores de red (Network Processors) en los routers y gateways de comunicación, en lugar de ASICs (Application-Specific Integrated Circuits) con funciones fijas, debido a que el último no tiene la flexibilidad necesaria para manejar la complejidad que implican los dispositivos de última generación. Intel está apostando fuertemente a esta tecnología y ofrece una línea completa de procesadores de red[1]. Estos componentes aprovechan, desde hace tiempo, la potencia de cálculo que ofrecen las arquitecturas CMP y SMP (por ejemplo el procesador IXP 1200 de Intel incluye 16 núcleos de procesamiento).

Entonces, asumiendo que tanto los clientes como los servidores y dispositivos de red estarán construidos sobre este tipo de arquitecturas, se han estudiado las formas de paralelizar el procesamiento de los protocolos de red de manera

[1] http://www.intel.com/design/network/products/npfamily/index.htm

eficiente. El software de un subsistema de comunicación generalmente consiste en las funciones de protocolos (como manejo de conexiones, control de flujo, segmentación/ensamblado de paquetes, etc) y los mecanismos del sistema operativo (como la administración de los threads/procesos, manejo de eventos asincrónicos, buffering, etc) que soportan la implementación y ejecución de las pilas de protocolos de comunicación. Ejecutar tanto las funciones de los protocolos como los mecanismos del sistema operativo en paralelo es una técnica que puede ayudar a incrementar la velocidad de procesamiento de red y reducir la latencia de comunicaciones.

Sin embargo para lograr mejoras notables de performance, el speed-up obtenido de la paralelización debe ser significativamente superior al overhead de sincronización y cambios de contexto asociados al procesamiento concurrente. Se han utilizado varias aproximaciones para escalar las implementaciones de las pilas de protocolos y poder explotar las arquitecturas disponibles. Las aproximaciones de paralelización usualmente incluyen un compromiso entre la posibilidad de balancear la carga entre múltiples procesadores y el overhead debido a la sincronización, scheduling, distribución de interrupciones, etc.

Desafortunadamente, las pilas de protocolos de red, en particular las implementaciones actuales de TCP/IP, son conocidas por su incapacidad de escalar bien en los sistemas operativos monolíticos de propósito general para arquitecturas tipo SMP. Es por este motivo que se han comenzado a estudiar las formas de mejorar la performance de red a través de diversas técnicas, como el uso de afinidad de procesos/threads e interrupciones a CPUs determinadas, para mejorar el comportamiento de cache.

En este capítulo, se describen brevemente las arquitecturas paralelas actuales de propósito general y se estudia una clasificación de los paradigmas de paralelización de protocolos. Finalmente, se identifica a uno de estos paradigmas como el modelo que adopta el kernel Linux 2.6, condicionado principalmente por la forma en que se distribuyen las interrupciones de hardware entre las CPUs de una máquina SMP y se mencionan los problemas de escalabilidad que implica este modelo.

5.2 Multiprocesamiento

Multiprocesamiento es un término genérico que indica el uso de dos o más CPUs en un mismo sistema de cómputo. Existen muchas variantes de este concepto básico dependiendo de la forma de implementarlo (múltiples núcleos en una oblea, múltiples chips en un módulo, etc).

En un sistema de multiprocesamiento todas las CPUs pueden ser tratadas iguales o algunas pueden utilizarse para propósitos especiales. Una combinación de decisiones de diseño del hardware y el sistema operativo determinan

la simetría (o falta de ella) en un sistema dado. Los sistemas que utilizan cada CPU de manera idéntica son llamados sistemas de multiprocesamiento simétrico (Symmetric Multiprocessing, o SMP). En los sistemas donde no todas las CPUs se tratan igual, los recursos se pueden dividir de varias maneras. Ejemplos de esto son los sistemas de multiprocesamiento asimétrico (asymmetric multiprocessing o ASMP) y los sistemas de multiprocesamiento con acceso a memoria no uniforme (non-uniform memory access multiprocessing o NUMA).

Los sistemas NUMA dedican diferentes chips de memoria a diferentes procesadores. Esto permite que los procesadores accedan a la memoria en paralelo, lo cual puede mejorar dramáticamente el rendimiento de memoria si los datos accedidos por un proceso están localizados en el banco de memoria cercano al procesador que está ejecutando ese proceso. Por otro lado, NUMA hace más costoso mover datos de un procesador a otro lo que significa, entre otras cosas, que el balanceo de carga es más costoso. Este modelo es común en supercomputadoras de gran escala. En los sistemas ASMP, los procesadores se especializan para tareas específicas. Por ejemplo, es posible que una CPU responda a todas las interrupciones del hardware, mientras el resto del trabajo en el sistema sea distribuido equitativamente entre el resto de las CPUs. O puede restringirse la ejecución del código en modo kernel a sólo un procesador (a uno específico o a un procesador a la vez), mientras el código en espacio de usuario se ejecuta en el resto de las CPUs. Actualmente, la alternativa más extendida es la de utilizar cualquier CPU de una máquina para realizar cualquier trabajo, aunque se está investigando activamente la aplicación de sistemas ASMP en diferentes dominios como, por ejemplo, en sistemas de tiempo real.

A continuación se mencionan las dos arquitecturas que serán la base de la mayoría de los servidores de Internet y estaciones de trabajo en un futuro cercano: Symmetric Multiprocessing y Chip Multiprocessing. No se pretende hacer un estudio detallado de las características del hardware, en cambio se intentan introducir los conceptos básicos y listar algunos ejemplos actuales de máquinas de este tipo.

Symmetric Multiprocessing

El multiprocesamiento simétrico implica una arquitectura de computadora donde dos o más procesadores idénticos se conectan a una única memoria principal compartida. La mayoría de los sistemas multiprocesadores actuales utilizan una arquitectura SMP. Los sistemas SMP permiten que cualquier procesador ejecute cualquier tarea sin importar donde se ubican los datos para esa tarea en la memoria. Con el soporte adecuado del sistema operativo, los sistemas SMP pueden balancear la carga entre los procesadores eficientemente. Por otro lado, las arquitecturas SMP profundizan el problema que representa la diferencia de velocidades entre la CPU y el acceso a los datos en memoria principal.

Actualmente el mercado de pequeños servidores y poderosas estaciones de trabajo está dominado por SMP. Los sistemas más populares están basados en los procesadores de Intel Xeon, Pentium D y Core Duo, o las series de procesadores de AMD Athlon64 X2 y Opteron. Otros procesadores no pertenecientes a la familia x86 que entran en este mercado son Sun Microsystems UltraSPARC, Fujitsu SPARC64, SGI MIPS, Intel Itanium, Hewlett Packard PA-RISC, DEC Alpha de Hewlett-Packard y la familia POWER y PowerPC de IBM. Los servidores de medianas prestaciones que utilizan entre cuatro y ocho procesadores se suelen basar en Intel Xeon MP, AMD Opteron 800 y los antes mencionados UltraSPARC, SPARC64, MIPS, Itanium, PA-RISC, Alpha y POWER. También existen sistemas de altas prestaciones con dieciséis o más procesadores de algunas de las series mencionadas anteriormente.

La llegada del multiprocesamiento simétrico permitió un nuevo grado de escalabilidad a los sistemas de cómputo. En lugar de derivar mayor performance de procesadores más rápidos, un sistema SMP combina múltiples procesadores para obtener grandes ganancias en la performance total del sistema. Suponiendo un nivel suficiente de paralelismo en el software, estos sistemas han probado poder escalar hasta cientos de procesadores.

Chip Multiprocessing

Más recientemente, un fenómeno similar está ocurriendo a nivel de chip. En lugar de profundizar los rendimientos decrecientes incrementando la performance individual de un procesador, los fabricantes están produciendo chips con múltiples núcleos de procesamiento en una única oblea. El multiprocesamiento a nivel de chip (conocido como Chip MultiProcessing o CMP) es multiprocesamiento simétrico implementado en un único circuito integrado VLSI. Los núcleos, además, pueden compartir un segundo o tercer nivel de caché común. Pueden mencionarse algunos ejemplos comerciales que se basan en este concepto:

- El POWER4 de IBM fue el primer procesador dual-core del mercado en el año 2000. Los chips dual-core POWER5 y PowerPC 970MP se usaron ampliamente en el mercado (el último, usado en la PowerMac G5 de Apple). Por otro lado, el procesador Cell es el resultado del esfuerzo conjunto entre IBM, Sony y Toshiba para desarrollar un procesador de última generación (actualmente utilizado en la consola de juegos PlayStation 3). Esta conformado por una arquitectura IBM POWER multicore y es uno de los procesadores de propósito general más potentes del mercado, aunque recién se están portando las aplicaciones y sistemas operativos a esta arquitectura. Linux es uno de los primeros sistemas operativos en ejecutarse sobre Cell.

- AMD lanzó sus procesadores dual-core Opteron para servidores y estacio-

nes de trabajo el 22 de Abril del 2005, y sus procesadores para escritorio, la familia Athlon 64 X2 el 31 de Mayo del 2005. AMD también lanzó recientemente el FX-60 y FX-62 para escritorios de alta performance y el Turion 64 X2 para laptops. AMD planea lanzar sus Opteron Quad-Core en el tercer trimestre de 2007.

- Intel vende sus procesadores Xeon dual-core a 3GHz, está desarrollando versiones dual-core de su arquitectura de alto rendimiento Itanium y ya produjo el Pentium D, la variante dual-core del Pentium 4. Por otro lado, los nuevos chips Core Duo y Core 2 Duo se incluyen en varias computadoras de Apple y en laptops de Sony, Toshiba, ASUS, etc. Intel ya presentó su primer procesador Quad-Core a principios del 2007.

- Ejemplos de Sun Microsystems son UltraSPARC IV, UltraSPARC IV+, UltraSPARC T1. Éste último, más conocido por su nombre código *Niagara*, es una CPU con SMT y multicore, diseñada para reducir el consumo de energía de los servidores y fue presentada en Noviembre del 2005. Se lo consideró el procesador comercial de propósito general más poderoso.

Las ventajas de una arquitectura CMP con respecto a SMP son varias. Entre otras se pueden incluir las siguientes:

- La cercanía de múltiples núcleos de CPU en la misma oblea tiene la ventaja de que la circuitería de coherencia de cache puede operar a una velocidad de reloj mucho más alta de lo que es posible en el caso SMP donde las señales deben viajar fuera del chip. Por lo tanto, se mejora la performance de las operaciones *snoop cache*.

- Físicamente, los diseños de CPU multicore requieren mucho menos espacio de la placa base que los diseños SMP.

- Un procesador dual-core usa menos electricidad que dos procesadores con un core cada uno, principalmente por el mayor poder requerido para manejar las señales externas al chip y porque su diseño permite que operen a menor voltaje. Además los núcleos comparten algunos componentes, como la cache L2 y la interfaz al front side bus.

5.3 Paralelización de protocolos

Suponiendo una gran disponibilidad de arquitecturas paralelas en el futuro cercano, el desafío es identificar el paralelismo en las implementaciones de los protocolos de comunicación. Es importante entender que el paralelismo en la implementación puede no ser visible al nivel de especificación (consideremos por

ejemplo la ejecución de transferencias DMA en paralelo con el procesamiento de paquetes).

El diseño de un subsistema de comunicación en una arquitectura paralela es actualmente un campo de investigación muy activo. Los problemas principales que se pueden mencionar son:

- Identificar el paralelismo: No es una tarea trivial ya que muchos protocolos fueron diseñados como una única máquina de estados. Para una implementación en paralelo, esta máquina de estados debe ser dividida en múltiples máquinas de estados que se comunican. Esto introduce overhead e incrementa la dificultad de probar formalmente su correctitud.

- Mapear threads a microprocesadores o a hardware dedicado: La discusión entre utilizar componentes de cómputo de propósito general o hardware VLSI dedicado para la ejecución de los protocolos no se ha cerrado, aunque en estos últimos años se ha frenado la tendencia a implementar funcionalidad *stateful* en hardware, principalmente por la falta de flexibilidad. Más allá de la elección del tipo de componente de cómputo utilizado, la forma de distribuir los threads de procesamiento de red a dichos componentes es de por sí un tema de estudio importante.

5.3.1 Formas de paralelismo

Un método bien conocido para clasificar el paralelismo es el de Flynn [27]. Sin embargo, esta categorización no alcanza para identificar el modelo de procesamiento de red en paralelo ya que, en general, la forma más utilizada de paralelismo en procesamiento de protocolos es *Multiple-Instruction, Multiple-Data (MIMD).* A continuación se ofrece una clasificación más detallada del paralelismo en el procesamiento de red [24]:

- Farming: Todos los procesadores ejecutan el mismo algoritmo sobre datos diferentes. Debe notarse que las tareas reales ejecutadas por un procesador pueden terminar siendo diferentes a las que realiza otro procesador. Por ejemplo, un paquete puede requerir checksuming mientras que otro paquete puede ser de control. Cuando se usa farming, el algoritmo del protocolo no es realmente desacoplado en sub-tareas. En su lugar el hardware es duplicado, se requieren mecanismos de sincronización y un scheduler (o algún otro componente de hardware o software) distribuye las tareas entre los procesadores.

- Paralelismo Temporal (Pipelining): El protocolo es dividido en procesos que son ejecutados secuencialmente en cada etapa del pipeline.

- Distribución Funcional (Multiple data parallelism): El protocolo es dividido en procesos principales independientes, como podrían ser un proceso de transmisión y un proceso de recepción.

- Single data parallelism: El algoritmo es dividido en varios procesos que operan sobre el mismo dato en paralelo. Esto es *Multiple-Instruction, Single-Data (MISD)* de acuerdo a la taxonomía de Flynn. Por ejemplo, cuando el dato es el encabezamiento de un paquete, las subtareas paralelizadas pueden ser la resolución de dirección, el chequeo del número de secuencia, etc.

5.3.2 Niveles de paralelismo

Las formas de paralelismo descriptas arriba pueden ser aplicadas a diferentes niveles, donde los niveles se distinguen por la granularidad de las tareas. Podemos identificar los siguientes niveles en el procesamiento de protocolos de red:

Basado en la tarea: Se asocian procesos[2] con grupos de una o más tareas del protocolo. Dos ejemplos representativos de esto son el paralelismo por capa y el paralelismo funcional. Cada capa puede involucrar múltiples tareas (por ejemplo, el reordenamiento es una tarea de la capa de transporte del modelo OSI).

El paralelismo por capas asocia un proceso separado con cada capa (por ejemplo, la capa de presentación, transporte y red) en una pila de protocolos. Ciertos encabezados y campos de datos en los mensajes salientes y entrantes pueden ser procesados en paralelo a medida que fluyen a través del pipeline formado por las capas de la pila de protocolos. Generalmente se necesita buffering y control de flujo ya que el procesamiento de cada capa puede darse a diferentes velocidades.

El paralelismo funcional asocia un proceso separado con cada función del protocolo (como creación del header, retransmisiones, segmentación, enrutamiento). Estas funciones se ejecutan en paralelo y se comunican enviándose los mensajes de datos y control del protocolo entre ellas.

En general, la implementación de una pila de protocolos como un pipeline es relativamente directa. El paralelismo basado en tareas se mapea directamente con los modelos en capas convencionales de comunicaciones utilizando diseños productor/consumidor. Incluso hay una necesidad mínima de mecanismos de sincronización entre las capas o funciones, ya que el procesamiento paralelo se serializa generalmente en el punto de acceso al servicio (como la interfaz de la capa de transporte, o de red). Sin embargo, como se ha mostrado en varios trabajos [24], este tipo de modelo es propenso a un gran overhead por context

[2] Aquí, el término proceso se utiliza para referir una serie de sentencias ejecutándose en un espacio de direccionamiento.

switching, principalmente cuando el número de tareas de protocolo excede el número de elementos de procesamiento y se producen cambios de contexto al transferir los mensajes entre las tareas.

Basado en Mensajes: asocia procesos con mensajes en lugar de capas o funciones de protocolos. Dos ejemplos comunes son paralelismo por conexión y paralelismo por mensaje. En el primer caso los mensajes de una misma conexión son procesados por un único proceso, mientras que en el segundo caso los mensajes se distribuyen entre cualquiera de los procesos disponibles.

El paralelismo por conexión utiliza un proceso separado para manejar los mensajes asociados con cada conexión. Para una conexión, un conjunto de tareas de protocolos son invocadas secuencialmente para cada mensaje a medida que éste fluye por la pila de protocolos.

El paralelismo por mensaje asocia un proceso separado con cada mensaje entrante o saliente. Un proceso recibe un mensaje de una aplicación o interfaz de red y realiza las tareas de los protocolos a través de la pila.

En general, existe un alto grado de paralelismo potencial a este nivel. Esto depende de características que cambian dinámicamente (como la cantidad de mensajes o conexiones), en lugar de depender de las características relativamente estáticas (como el número de capas o funciones de protocolos) que se asocian con el modelo basado en tareas.

5.4 El kernel Linux 2.6 en arquitecturas SMP

A continuación se estudian las características de Linux 2.6 cuando se ejecuta en una arquitectura SMP. Debido a la forma en que se distribuyen las interrupciones por hardware y la manera de planificar las softirqs de red, se llega a la conclusión de que este kernel sigue el paradigma de paralelismo por mensaje. Pero también se observa que este modelo no presenta un buen comportamiento con respecto a la escalabilidad.

5.4.1 El Controlador programable avanzado de interrupciones de E/S

Cada controlador de dispositivo de hardware capaz de elevar interrupciones suele tener una única línea de salida designada como la línea de requerimiento de interrupción (IRQ o Interrupt ReQuest). Los dispositivos más sofisticados pueden usar varias líneas IRQ (una placa PCI puede usar hasta cuatro líneas IRQ). Todas las líneas IRQ existentes se conectan a los pines de entrada de un circuito llamado Controlador Programable de Interrupciones (PIC o Programmable Interrupt Controller). El esquema monoprocesador tradicional conectaba en cascada dos PICs (chips del tipo 8259A) y la línea de salida del PIC master se conectaba directamente al pin INTR de la CPU.

Sin embargo en sistemas multiprocesador, se necesita una solución algo más compleja. Es necesario poder enviar interrupciones a todas las CPUs para aprovechar el paralelismo de las arquitecturas SMP. Por esta razón, desde el Pentium III, Intel introdujo un nuevo componente designado como el Controlador Programable Avanzado de Interrupciones de E/S (I/O APIC o I/O Advanced Programmable Interrupt Controller). Este chip es la versión avanzada del anterior PIC tipo 8259A.

Además, los procesadores actuales de la familia 80x86 incluyen un APIC local que contiene registros de 32 bits, un reloj interno, un timer y dos líneas IRQ adicionales, LINT0 y LINT1, reservadas para interrupciones de APIC locales. Todos los APICs locales se conectan al I/O APIC externo creando un sistema multi-APIC.

La figura 5.1 ilustra de manera esquemática la estructura de un sistema multi-APIC. Un bus conecta el I/O APIC con los APICs locales. Las líneas IRQ provenientes de los dispositivos son conectadas al I/O APIC, el cual actúa como router con respecto a los APICs locales.

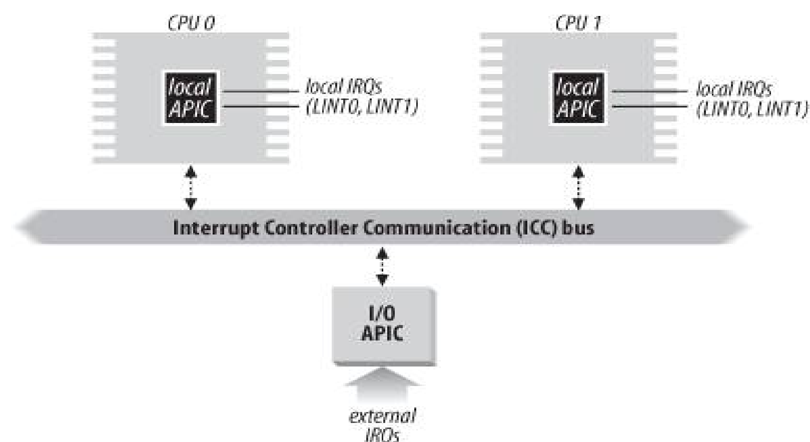

Figura 5.1: Configuración multi-APIC con dos procesadores [7]

El I/O APIC es el responsable de distribuir las interrupciones de dispositivos de hardware a los APICs locales en los procesadores [7], para lo cual incluye 24 líneas IRQ, una tabla de redireccionamiento de interrupciones de 24 entradas, registros programables y una unidad para el envío y recepción de mensajes sobre el bus APIC. Cada entrada en la tabla puede ser programada individualmente para indicar el procesador destino.

Dependiendo de la configuración programada en el I/O APIC, las interrupciones son o bien distribuidas estáticamente a los procesadores especificados en la tabla de redirección del controlador, o dinámicamente al procesador que esta

ejecutando el proceso con la prioridad más baja.

- Distribución Estática: La señal IRQ es despachada al APIC local listado en la entrada correspondiente en la tabla de redirección. La interrupción puede ser enviada a una única CPU, un subconjunto de CPUs o a todas las CPUs del sistema (modo broadcast).
- Distribución Dinámica: La señal IRQ es despachada al APIC local del procesador que esté ejecutando el proceso con la prioridad más baja. Cada APIC local tiene un registro de prioridad de tarea (TPR o Task Priority Register), el cual es usado para computar la prioridad del proceso que se está ejecutando en un determinado momento. Se espera que este registro sea actualizado por el sistema operativo en cada cambio de proceso. Si dos o más CPUs comparten la prioridad más baja, la carga se distribuye entre ellas usando una técnica llamada arbitración, que logra un esquema round-robin.

Además de distribuir interrupciones entre procesadores, un sistema multi-APIC permite que las CPUs generen interrupciones entre procesadores. Éstas son un componente fundamental de la arquitectura SMP y son muy usadas por Linux para intercambiar mensajes entre CPUs.

Tanto Windows NT como Linux 2.4, por defecto se configuraban para operar en el modo de prioridad más baja. Desafortunadamente, en algunos casos el hardware fallaba en distribuir las interrupciones entre los procesadores de manera equitativa (por ejemplo, algunas motherboards SMP de 4 CPUs tenían este problema) y esto causaba que las interrupciones de dispositivos se enviaran a la CPU0. Durante cargas altas, la CPU0 se saturaba antes que otros procesadores en el sistema, llevando a un cuello de botella.

En Linux 2.6 se implementa un esquema más inteligente, donde el kernel despacha interrupciones a un procesador por un pequeño período y luego cambia aleatoriamente el envío de interrupciones a un procesador diferente, para esto se hace uso de un thread especial del kernel llamado kirqd.[3] Modificando las entradas de la tabla de redirección de interrupciones es posible enrutar la señal de una interrupción a una CPU específica. El thread kirqd mantiene un registro del número de interrupciones recibidas por cada CPU en un período de tiempo. Si el desbalance entre la CPU más cargada y la menos cargada es significativo, entonces hace una rotación de IRQs entre las CPUs existentes.

Esta distribución aleatoria y periódica resuelve el problema del cuello de botella del sistema (CPU0) y provee una aproximación de mejor esfuerzo al tema de la afinidad de interrupciones a CPUs. Sin embargo, los intentos no coordinados de distribuir las interrupciones a diferentes procesadores puede resultar

[3]Se continúa trabajando en este mecanismo para lograr un esquema de distribución de interrupciones más inteligente y menos aleatorio.

en efectos secundarios indeseables. Los manejadores de interrupciones terminan siendo ejecutados en procesadores aleatorios, lo que crea más contención para recursos compartidos, y los contenidos de las memorias cache (el contexto de una conexión TCP, por ejemplo) no son reusados óptimamente ya que los paquetes de una misma conexión pueden ser enviados a procesadores diferentes en cada interrupción de las NICs.

5.5 Problemas de Escalabilidad de TCP/IP en Linux

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 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 el capítulo 8, 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". Es decir, cuando se agregan CPUs adicionales al procesamiento de red, el beneficio parece crecer linealmente pero el costo de contención, serialización, scheduling, etc., crece geométricamente.

Como se dijo anteriormente, 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 [20], 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 las 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 interfaz de red comienza la tarea de recibirlo. El procesamiento central de red sucede en la softirq net_rx_action(). 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 de algún modo. 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 de sincronización.

- El código de red 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 ubicaciones en memoria. Ya que muchos componentes del subsistema de red tienen una relación productor/consumidor, una cola FIFO libre de locks puede realizar esta función con mayor eficiencia.

Jacobson afirma que la clave para una mayor escalabilidad de red es liberarse del locking y de los datos compartidos tanto como sea posible. Sin embargo, siempre que se continúe 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, los costos incurridos por interrupciones y scheduling tienen un impacto importante en la efectividad de la cache. El scheduler en sistemas operativos típicos, 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 migración de proceso, el proceso migrado debe alimentar varios niveles de caches de datos en el procesador al cual fue migrado.

5.5.1 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 ejecute 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 y NICs, pueden ser apropiados para este tipo de mecanismos. Dos servidores web comúnmente usados (TUX de Red Hat, e 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 maquinas SMP/CMP, extendiendo estas ideas más allá del procesamiento de red.

Capítulo 6

Reduciendo la intrusión del sistema operativo

6.1 Introducción

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.

Claramente, existe la necesidad de especializar los sistemas operativos. No resulta lógico utilizar el mismo kernel en una máquina de escritorio, donde el time sharing y el tiempo de respuesta percibido por el usuario son factores esenciales, que en un nodo de almacenamiento de una infraestructura Grid de donde tal vez se transfieran cientos o miles de Gigabytes de datos por día, desde el disco o cinta a la red y viceversa.

Existe un conjunto de proyectos que han reconocido la necesidad de rediseñar los sistemas operativos para adaptarlos a los nuevos requerimientos y necesidades. La motivación para diseñar un nuevo sistema operativo proviene de dos factores independientes: las deficiencias en el uso de los SO de propósito general existentes para aplicaciones de red y E/S en general, y la creciente disponibilidad de variantes de multiprocesadores. Un sistema operativo correctamente diseñado, debería incrementar los beneficios de las plataformas multiprocesador en el procesamiento de red.

La gran deficiencia generada al usar sistemas operativos de propósito general como la base para servicios de red surge de no tener en cuenta las características

del trabajo que realizan estos nodos la mayor parte del tiempo. En general los servidores principalmente mueven datos (en lugar de crearlos o procesarlos) ya sea, desde el almacenamiento a la red (servidores) o desde la red a la red (routers y otros dispositivos de red). Podemos caracterizar el trabajo de un servidor como predominantemente de transferencia de datos y, por lo tanto, las operaciones realizadas por el software serán obviamente diferentes a aquellas de un sistema usado para tareas de computación más generales. Por lo tanto parece razonable asumir que un sistema operativo especializado para esa tarea de movimiento de datos puede ser más eficiente que un sistema más general.

Por otro lado, un sistema operativo que es diseñado específicamente para correr en sistemas multiprocesador debería hacer un mejor uso de los procesadores que un sistema operativo monoprocesador adaptado a un entorno multiprocesador, como lo son la mayoría de los sistemas operativos SMP actuales.

6.2 Intrusión del Sistema Operativo

El objetivo principal de un sistema operativo moderno es el de proveer la ejecución concurrente de múltiples programas independientes. Esto es equivalente a la multiplexación de los recursos compartidos entre esos programas, una tarea usualmente realizada a través de la virtualización de esos recursos, consistente en las siguientes funciones interrelacionadas:

- Protección: prevenir que las aplicaciones accedan a los recursos en modos diferentes a los cuales fueron autorizadas.
- Multiplexación: proveer virtualizaciones independientes de un único recurso físico a múltiples aplicaciones.
- Traducción: mapear la instancia virtualizada utilizada por cada aplicación con el recurso físico subyacente.

Por ejemplo, consideremos la memoria virtual. Las tablas de páginas proveen la traducción desde direcciones virtuales hacia la memoria física multiplexada y son protegidas de accesos no autorizados por parte de las aplicaciones.

Tradicionalmente estas primitivas de virtualización fueron innecesariamente combinadas con las funciones de alto nivel provistas por el sistema operativo, como sistemas de archivos y pilas de protocolos de red. Esta combinación proviene de lo que históricamente fue un rol importante del sistema operativo: proveer una librería común de funciones que podía ser usada por los programas que se ejecutaban en una máquina. Sin embargo, la combinación entre las tareas de virtualización y las funciones de alto nivel pueden llevar a comportamientos no óptimos por parte de las aplicaciones y el sistema operativo. Por lo tanto,

actualmente existe gran interés en desacoplar ambos roles. Hay proyectos importantes de investigación que siguen este camino como son el kernel Nemesis de la Universidad de Cambridge[1] y el Exokernel del MIT[2].

Este comportamiento no óptimo es el generador del concepto conocido como intrusión del sistema operativo. En términos generales, se define la intrusión del sistema operativo como el overhead generado por éste al cumplir sus funciones de protección, virtualización, y traducción. Se puede afirmar que la intrusión tiene una relación directa con la manera en que las políticas y mecanismos del sistema operativo se adaptan a la forma en que las aplicaciones lo utilizarán. Existe una regla práctica para determinar cuándo el uso de un recurso por parte del sistema operativo debe ser considerado intrusión o no: si se evalúa una aplicación que se ejecuta con acceso directo a todos los recursos físicos y consume la misma cantidad de recursos, entonces el uso del recurso por parte del sistema operativo no se considera intrusión.

Podemos separar a la intrusión en dos categorías: intrusión por recursos e intrusión por tiempo. La última se subdivide a su vez en intrusión por política e intrusión por mecanismo.

Intrusión por recursos

Denota el uso de recursos físicos (por ejemplo, memoria, bloques de disco, ancho de banda de la red) por parte del sistema operativo para sus propios objetivos, es decir, para propósitos que no ayudan al progreso de la aplicación. Por ejemplo, los bloques de disco usados para almacenar metadatos (como inodos en un filesystem UNIX). Aunque en este trabajo no se estudia este tipo de intrusión, es un campo de investigación importante y activo.

Intrusión por política: La optimización para el caso general puede ser demasiado costosa

La intrusión por política sucede cuando las funciones provistas por un sistema operativo incluyen políticas que tienen efectos perjudiciales para el comportamiento de la aplicación.

Un ejemplo representativo de estas decisiones de políticas es el del *read ahead*. El read ahead es realizado por el sistema operativo para incrementar la eficiencia en el acceso a los dispositivos de almacenamiento. Generalmente se leen a memoria múltiples bloques del disco para amortizar el overhead asociado con el requerimiento de un único bloque de datos. La política determina cuantos bloques adicionales deberían leerse a memoria, si la aplicación requiere un número menor que el mínimo. El read ahead secuencial (leer, del disco a memoria, más

[1] http://www.cl.cam.ac.uk/research/srg/netos

[2] http://pdos.csail.mit.edu/exo.html

bloques que los solicitados por la aplicación) es la política más común en los sistemas operativos modernos. Sin embargo esta técnica no es apropiada, por ejemplo, para los DBMSs, que suelen realizar accesos aleatorios a los archivos. Por lo tanto el sistema operativo desaprovecha, en ese caso, ancho de banda del bus y tiempo leyendo bloques que nunca serán utilizados. Este ejemplo es uno de los primeros casos notables de intrusión por política que motivaron la búsqueda de alternativas.

Intrusión por mecanismo: El precio de la seguridad

Para soportar la primera de las primitivas de virtualización de los recursos, es decir la protección, un sistema operativo debe aislar los recursos físicos de las aplicaciones que los usan. De otra forma, una aplicación maliciosa con acceso directo a los recursos podría violar la multiplexación que realiza el sistema operativo, bien para obtener más recursos o bien para interferir con los recursos asignados a otras aplicaciones.

La protección es casi universalmente implementada particionando el dominio de ejecución en regiones privilegiadas y no privilegiadas, que suelen denominarse como nivel supervisor y nivel usuario. El sistema operativo se ejecuta en nivel supervisor, con acceso a todas las capacidades e instrucciones de la CPU, mientras que las aplicaciones se ejecutan en nivel usuario, y sólo acceden a un subconjunto seguro de instrucciones.

Una vez que el kernel ha erigido los límites de privilegios alrededor suyo para proveer protección de los recursos, se vuelve necesario cruzar esos límites cuando se necesita invocar los servicios del kernel. Esto puede ocurrir en dos situaciones:

- Cuando una aplicación desea invocar un servicio del sistema operativo. Por ejemplo, escribir datos en un archivo.
- Cuando un dispositivo de hardware eleva una interrupción para requerir que el driver maneje algún cambio de estado en el dispositivo. Por ejemplo, el arribo de un paquete de red.

Aunque a ambas ocurrencias se las puede referir como interrupciones, a la primera se las suelen llamar interrupciones sincrónicas o traps y a la segunda interrupciones asincrónicas. Estos eventos generan dos tipos distintos de intrusión y aunque comparten ciertas similitudes, también hay diferencias importantes entre ellas. Se las conoce como: la intrusión sincrónica y la intrusión asincrónica.

- Intrusión sincrónica: La intrusión sincrónica ocurre cuando una aplicación ejecuta una instrucción de trap, la cual causa que la CPU cambie de un dominio de ejecución no privilegiado a uno privilegiado. En este caso,

el estado que debe ser guardado puede ser fácilmente controlado por el sistema operativo. Esta situación es análoga a una llamada a función, en el sentido de que es el código llamador el que que debe guardar su estado en el caso que se quiera preservarlo. Otras tareas que pueden incrementar el costo de las traps incluyen el actualizar los registros de control de la memoria virtual, cambiar a la pila en modo kernel, etc.

- Intrusión asincrónica: El overhead de una interrupción es mucho mayor que el de un trap por dos razones: primero, la naturaleza asincrónica de la interrupción significa que la aplicación se encuentra en un estado desconocido cuando ésta ocurre, por lo que el sistema operativo debe actuar de manera conservadora, preservando todo el estado que podría ser afectado por el manejo de las interrupciones. Segundo, la señalización utilizada para elevar y responder a una interrupción usualmente requiere que ocurran varias transacciones del bus.

Tanto las interrupciones como los traps tienen efectos adversos más allá que la penalidad inmediata del tiempo, por ejemplo los costos por scheduling y la polución de cache generada.

Intrusión relacionada e intrusión no relacionada

Cuando se evalúa la intrusión, es importante tener en cuenta en qué grado la intrusión está relacionada con el comportamiento de la aplicación. La intrusión que está directamente asociada con las acciones realizadas por las aplicaciones se conoce como intrusión relacionada. La intrusión no relacionada, por otro lado, ocurre sin relación con las acciones de la aplicación.

La intrusión sincrónica esta completamente relacionada ya que son invocaciones al kernel por parte de la aplicación. La intrusión asincrónica puede ser relacionada, por ejemplo una interrupción debido a un paquete de red recibido como respuesta a un mensaje previo enviado por la aplicación, o totalmente desasociada, por ejemplo las interrupciones del timer.

Intrusión en sistemas multiprocesador

Los sistemas multiprocesador son particularmente propensos a la intrusión por mecanismo, ya que la mayoría de los sistemas operativos de propósito general que se usan en estas arquitecturas fueron adaptados desde una base monoprocesador. Por lo tanto, además de tener que coordinar los accesos de múltiples aplicaciones independientes a un único recurso físico, el sistema operativo en un sistema multiprocesador también debe propagar esta protección entre múltiples CPUs. La sincronización necesaria introduce una capa extra de intrusión por mecanismo.

Consideremos un sistema que serializa los accesos a los recursos del kernel protegiendo el kernel entero con una única variable de sincronización, como en el caso del *Big Kernel Lock* introducido a partir de la versión de Linux 2.0. Si varias aplicaciones pasan una fracción significativa de su tiempo de ejecución en el kernel, como debe esperarse de una aplicación de transferencia de datos, entonces se pierde toda ventaja de la arquitectura paralela, ya que las aplicaciones serán serializadas por el lock global y el sistema multiprocesador. De hecho, la performance podría ser peor, por el control extra de la concurrencia. Afortunadamente, las invocaciones a este lock prácticamente han desaparecido del código del kernel Linux y en su lugar se utilizan locks de granularidad mucho más fina.

Factor de Intrusión y un caso paradigmático

La intrusión del sistema operativo puede cuantificarse de la siguiente forma

`FI` = tiempo real de ejecución / tiempo ideal de ejecución

donde el tiempo real e ideal de ejecución son respectivamente el tiempo de ejecución de una operación dada con y sin un sistema operativo presente. Obviamente, un sistema operativo ideal tendría un factor de intrusión (FI) mínimo de 1, para todas sus operaciones.

Un caso particularmente interesante es cuando el FI tiende a infinito, es decir, aunque se ejecute código del sistema operativo, no se realiza ningún trabajo real para el progreso de la aplicación. Este caso fue estudiado en algunos trabajos y se lo suele denominar con el término de *Livelock*. El clásico ejemplo se conoce como Receive Livelock debido al procesamiento de interrupciones de red de tramas entrantes, como lo describe Ramakrisan en su importante publicación [17]. En su trabajo, se observó que un kernel UNIX no hacía ningún progreso al ejecutar aplicaciones porque el 100 % del tiempo de CPU era consumido por el manejo de interrupciones relacionadas con tramas entrantes, que luego eran desechadas por el propio kernel por no disponer de recursos de CPU suficientes para su procesamiento.

El FI relativamente alto medido en varios sistemas operativos / arquitecturas demuestra que el costo de la intrusión es realmente significativo si la frecuencia de la intrusión es lo suficientemente alta. Por lo tanto, uno de los objetivos principales del diseño de un sistema operativo para servicios de red es la minimización de los costos de la intrusión por mecanismo.

6.3 La evolución de los sistemas operativos en búsqueda de menor intrusión

Cuando las computadoras eran costosas y lentas las prioridades de los sistemas operativos eran maximizar la disponibilidad del hardware para muchos usuarios. Ejemplos de esto son los sistemas de time-sharing como *Compatible Time-Sharing System* (CTSS) y *Multics* que soportaban múltiples aplicaciones de usuario por máquina, y los sistemas de máquina virtual como *IBM VM/370* que soportaban múltiples sistemas operativos en una única máquina física.

Sin embargo, en el principio de la década de 1980, la aparición de estaciones de trabajo de alta performance cambiaron el mundo de la computación y, por lo tanto, los requerimientos de los sistemas operativos. Acompañando este cambio, se incrementó la presencia de los sistemas operativos UNIX. Por la disponibilidad del código fuente de la implementación *Berkeley System Distribution* (BSD), muchos vendedores desarrollaron variantes de UNIX: HP-UX, Solaris, Ultrix, Irix. Por lo tanto UNIX y sus derivados rápidamente se transformaron en el sistema operativo predominante para estaciones de trabajo de alta performance y servidores.

6.3.1 Kernel monolítico

Unix y sus derivados comparten la arquitectura mostrada en la figura 6.1, usualmente referida como kernel monolítico. El kernel se ejecuta en modo privilegiado mientras que las aplicaciones se ejecutan en modo no privilegiado, previniendo de esta forma el acceso directo a los recursos físicos. El kernel provee servicios de alto nivel a las aplicaciones, por ejemplo las pilas de protocolos de red y sistemas de archivos, como el único mecanismo por el cual aquellas pueden hacer uso de los recursos físicos. Por lo tanto la virtualización es realizada a través de la abstracción, escondiendo todos los detalles de la administración de los recursos a las aplicaciones.

Tradicionalmente, la mayor parte de la pila de protocolos de red es incluida como parte de un kernel monolítico. Los fundamentos de esta decisión tienen que ver principalmente con razones de políticas y performance. Las políticas claves en este caso son fairness (por ejemplo al multiplexar los flujos de paquetes) y la prevención de inanición. El código de red puede requerir la habilidad de controlar el timing y el scheduling de los threads asociados, manipular memoria virtual directamente y controlar completamente los dispositivos relacionados, etc. Además, es natural colocar los drivers de dispositivos bajo la protección del sistema operativo. Los sistemas UNIX, como Linux, BSD y Solaris están en esta categoría.

En estos años se ha comenzado a cuestionar la conveniencia de éste modelo de funcionamiento. “La forma en que siempre fue hecho no es necesariamente

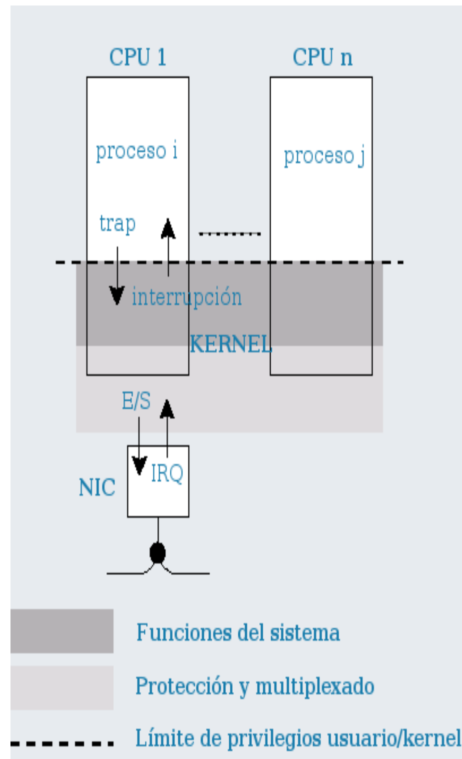

Figura 6.1: Kernel Monolítico

la mejor forma de hacerlo", afirma Van Jacobson en su última propuesta [8], insinuando que las decisiones históricas en el diseño de la pila TCP/IP deberían ser revisadas.

Multics, creado por ARPA, incluía la primer pila de red (ARPAnet) que se implementó en un sistema operativo (MIT, 1970). Corría en una *supercomputadora* multiusuario (de 0.4 MIPS) y raramente había menos de 100 usuarios conectados. Tardaba cerca de 2 minutos en realizar la operación de paginación, motivo por el cual la pila de protocolos tuvo que ser incluida en el kernel, ya que la memoria del kernel no paginaba.

Hacia 1980 se implementó la primer pila TCP/IP en Multics y, posteriormente, gente del mismo grupo de trabajo se trasladó a BBN (Bolt, Beranek, and Newman, histórica empresa fundada por académicos del MIT) para implementar la primer pila TCP/IP para el Unix de Berkeley (BSD) en 1983. El Computer Systems Research Group (CSRG), en Berkeley, usó la pila de BBN como especificación funcional para la pila de BSD 4.1 en 1985 y luego el código fue liberado bajo la licencia BSD en 1987.

Por lo tanto, la arquitectura Multics se convierte en el modelo estándar de diseño de la pila de protocolos más utilizada. Es difícil exagerar la influencia que tuvo esta implementación en el resto de las implementaciones de TCP/IP a lo largo de la historia, incluyendo por supuesto, a la del kernel Linux. Sin embargo, como se mencionó previamente, se han comenzado a evaluar las decisiones de diseño heredadas de la arquitectura Multics.

6.3.2 El microkernel

Un kernel monolítico ofrece varias ventajas para aplicaciones de propósito general. Los defensores de este modelo citan la simplicidad de implementación y performance como principales ventajas. La comunicación dentro del kernel es sencilla y eficiente ya que todo el código se ejecuta en el mismo espacio de direccionamiento, básicamente se invocan funciones.

Sin embargo, rápidamente se notó que tal estructura de kernel era muy restrictiva para ciertas aplicaciones. Específicamente, cuando las abstracciones provistas por el kernel no eran compatibles con los requerimientos de una aplicación, se disminuían notablemente la eficiencia y la performance.

El primer intento de resolver este problema fue mover los servicios del sistema operativo fuera del kernel, a procesos de usuario usualmente llamados *servers*. Esta aproximación se conoce con el nombre de microkernel. En este modelo, el kernel implementa sólo un mínimo de servicios de virtualización, delegando la provisión de servicios, como el de red y sistemas de archivos, a procesos server. Las aplicaciones requieren servicios de los servers enviándoles mensajes usando primitivas de comunicación entre procesos (InterProcess Communication o IPC) provistas por el kernel. Al soportar los servicios en espacio de usuario, el sistema operativo puede ofrecer diferentes estrategias de administración de recursos a diferentes aplicaciones, evitando el problema de la poca eficiencia de configuración genérica en determinados casos.

Sin embargo, se ha afirmado que este tipo de diseño es intrínsecamente inefectivo en soportar redes de alta velocidad [19]. En un sistema operativo basado en microkernel, diferentes partes del subsistema de red residen en dominios de protección diferentes, lo que significa que los requerimientos de E/S y los datos asociados deben cruzar límites de protección y puntos de scheduling adicionales. Serán necesarios más transferencias de datos entre dominios y más cambios de contexto que en un sistema monolítico, dando como resultado una mala localidad de las referencias, un incremento en la carga del bus a memoria, una reducción del rendimiento de la aplicación y un incremento en la latencia de la red. Y no sólo se presenta este problema en el procesamiento de red, en general, los obstáculos mencionados estarán presente en cualquier procesamiento de E/S.

Además, mientras la aproximación de microkernel prometía flexibilidad en la administración de recursos, las primeras implementaciones se caracterizaron por

los altos overheads del IPC. Aunque sistemas posteriores, como la familia L4[3], redujeron notablemente este overhead, otros grupos propusieron una solución alternativa conocida como los kernels verticales.

6.3.3 Kernel verticalmente estructurado

Los diseñadores de kernels verticales mencionan dos fallas en el diseño de micro-kernel: el overhead del IPC y los problemas de scheduling causados por ejecutar todo el código de servicios en servers en espacio de usuario. Más específicamente, cuando una aplicación envía un requerimiento a un server, es necesario transferirle las propiedades de scheduling de la aplicación (como la prioridad) al server, y el uso de los recursos que implica la ejecución del servicio invocado debe ser correctamente contabilizado y sumado al registro del uso de recursos de la aplicación.

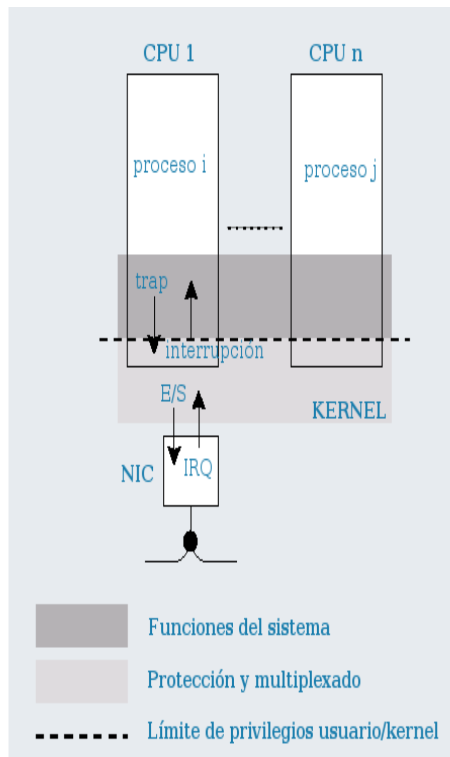

Figura 6.2: Kernel Vertical

En los sistemas estructurados verticalmente, como se muestra en la figura 6.2,

[3]http://www.l4hq.org/

el kernel aún provee las primitivas de virtualización de bajo nivel. Sin embargo, en contraste a un microkernel, el kernel también provee a las aplicaciones con un acceso directo, aunque restringido, a los recursos físicos, usando funciones de bajo nivel (es decir, no se invocan funciones de un proceso server). De esta forma, las aplicaciones aún retienen la habilidad de implementar sus propias funciones de alto nivel de administración de recursos, pero sin tener que actuar a través de un intermediario. Por lo tanto se elimina el overhead del IPC y el uso de recursos puede ser contabilizado más fácilmente. Por supuesto, las implementaciones de propósito general de muchos servicios serán suficientemente apropiadas para la mayoría de las aplicaciones, por lo tanto, las librerías compartidas son una parte integral de este tipo de sistema para proveer las funciones comunes a la aplicaciones sin requerimientos especiales.

6.3.4 El próximo paso: un kernel activo

Los sistemas verticales ofrecen la mayor separación entre las primitivas de virtualización y las funciones de servicio de alto nivel, y de esta forma minimiza la intrusión por política. Pero falla en solucionar uno de los problemas que motivan la necesidad de sistemas operativos específicos para servicios de red: el overhead de la intrusión por mecanismo.

En cada una de las arquitecturas de sistemas operativos anteriores, el kernel es un objeto pasivo. Esencialmente, es una librería dinámica con propiedades semánticas especiales (se ejecuta en un dominio de ejecución privilegiado). Pero los mecanismos usados para forzar esta semántica generan un costo asociado a las aplicaciones. La observación de que este overhead es significativo en un sistema donde las aplicaciones deben cruzar frecuentemente el limite de privilegios nos llevan a una conclusión obvia: es necesario evitar el alto costo de cruzar los límites de privilegios.

Un kernel activo permite la comunicación con las aplicaciones usando canales de comunicación implementados con memoria compartida. Las aplicaciones envían mensajes requiriendo que se ejecute una función en particular, el kernel obtiene el mensaje y procesa el requerimiento. Esta solución propone un kernel que se ejecute continuamente. La primer implementación de un kernel de este tipo se denominó Piglet [19], mostrado en la figura 6.3.

La anatomía de Piglet, el primer kernel activo

Gran parte de la arquitectura de Piglet es similar a un sistema estructurado verticalmente, en particular se destaca la separación de las primitivas de virtualización provistas por el kernel y las funciones de alto nivel implementadas directamente por las aplicaciones y librerías compartidas. Piglet, sin embargo es diferente en el mecanismo por el cual las operaciones de bajo nivel son in-

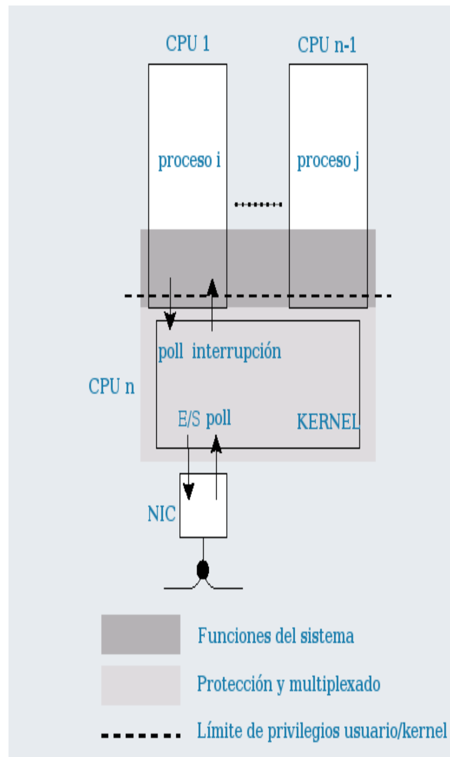

Figura 6.3: Kernel Activo

vocadas por las aplicaciones. En lugar de ejecutar un trap para cruzar el límite de privilegios y ejecutar código del kernel, se envía un mensaje en una ubicación en memoria compartida y el kernel lo recupera y lo procesa. Estas son las características fundamentales de Piglet:

- La decisión fundamental en el diseño de Piglet es que el kernel corra continuamente en un procesador dedicado, por lo tanto fue implementado para sistemas multiprocesador. Esto es necesario para eliminar las interrupciones y poder usar el modelo de comunicación de memoria compartida.

- La segunda decisión importante incorporada a Piglet es la eliminación de las interrupciones asincrónicas. Se presentan dos motivaciones: primero, el overhead del mecanismo de interrupciones es realmente alto cuando se realiza gran cantidad de E/S; segundo, las interrupciones asincrónicas agregan una complejidad significativa al sistema operativo. Esta complejidad se genera de la necesidad de actualizar atómicamente las estructuras de datos internas para garantizar que el estado del kernel se mantiene

consistente. La aproximación más común para hacer esto es deshabilitar las interrupciones mientras se ejecutan estas regiones críticas, sin embargo esta aproximación incrementa la probabilidad de errores por parte del programador y puede aumentar la latencia de las interrupciones.

- La última decisión de diseño, pero no menos importante, es la de utilizar objetos en memoria compartida como el mecanismo principal de comunicación entre las aplicaciones y el kernel. Ésto elimina el overhead del esquema tradicional, ya que los objetos se encuentran mapeados directamente en el espacio de direcciones de la aplicación. El kernel activo facilita este modo de comunicación porque los objetos compartidos pueden ser consultados con suficiente frecuencia. Un buen diseño del canal de comunicación es importante para mantener bajo el overhead de este mecanismo. La estructura de datos más apropiada es una cola FIFO usando métodos de sincronización no bloqueantes. El diseño de la cola permite que sea accedida por multiples lectores sin control de concurrencia, pero un único escritor a la vez puede accederla.

Aunque Piglet es sólo un prototipo, sus ideas fundamentales están siendo adoptadas por desarrollos comerciales que prometen revolucionar el codiseño hardware/software para las arquitecturas de entrada/salida de alta performance. El proyecto de Intel denominado IOAT, estudiado en el próximo capítulo, es un ejemplo claro de la viabilidad técnica de estas ideas.

Capítulo 7

Proyectos Relacionados

7.1 Introducción

Como se vio anteriormente, múltiples interrupciones de hardware, pilas de protocolos de red implementadas de manera ineficiente, demasiados accesos a memoria, un uso ineficiente de caches de datos e instrucciones y un overhead significativo agregado por el sistema operativo, consumen una cantidad de ciclos de CPU por paquete demasiado alta. Esto empeora a medida que se incrementa la velocidad de las comunicaciones: A 10 Gbps, los requerimientos de procesamiento superan el poder de cómputo de las plataformas actuales más utilizadas para servidores [10, 6].

Los investigadores son conscientes de que estos factores requieren cambios importantes en el co-diseño hardware/software de los servidores de red y han comenzado a proponer algunas soluciones para atacar diferentes aspectos de los mecanismos usados en el procesamiento de red. Existe un conjunto de proyectos, tanto académicos como comerciales, que intentan mitigar los factores mencionados anteriormente. Algunas soluciones se basan exclusivamente en modificaciones orientadas al hardware y otras sólo proponen nuevos diseños de software. Sin embargo, la mejor solución se presenta como una combinación de nuevas tecnologías de hardware con importantes cambios del sistema operativo.

Durante años, las aplicaciones con gran sensibilidad a la latencia de red y requerimientos específicos de alto rendimiento han adoptado TCP/IP sobre enlaces de diferentes velocidades. Para hacer que TCP/IP responda mejor en estos escenarios, los investigadores han evaluado diversas mejoras para el procesamiento de los protocolos. Inicialmente, la principal fuente de overhead en el procesamiento TCP/IP no había sido identificada completamente, y los desarrolladores asumieron que el culpable era el procesamiento inherente al fun-

cionamiento de los protocolos. Sin embargo, más adelante se demostró que el procesamiento específico de TCP no era el principal generador de overhead por sí solo; la fuente de mayor overhead era el entorno en el que se ejecutaba este protocolo: interrupciones, scheduling, buffering, movimiento de datos, etc.

Para solucionar algunos de estos problemas, los investigadores desarrollaron varios mecanismos que son comunes en las plataformas y pilas TCP/IP actuales. A continuación se estudian algunos de ellos.

7.2 Evolución del procesamiento de red, las primeras optimizaciones

En esta sección, se analizan algunas de las mejoras introducidas en los últimos años en el hardware y software involucrado en TCP/IP. La mayoría de ellas ya se incluyen por defecto en las NICs que se ofrecen en el mercado y son soportadas por los sistemas operativos y drivers.

Fueron pensadas principalmente para acompañar a la transición de ancho de banda desde los cientos de Megabits por segundo al Gigabit por segundo. Sin embargo, estas técnicas demostraron no ser suficientes para lograr un procesamiento eficiente de varios Gigabit por segundo. Por lo que ya existen nuevas propuestas para optimizar el tratamiento de paquetes de red.

Debe aclararse que no se estudian las extensiones que se produjeron a los algoritmos a nivel de especificación. Por ejemplo, TCP ha sido extendido y mejorado varias veces en la historia de Internet. Sin embargo, las mejoras que estudiamos en este capítulo no suelen contemplarse en la especificación del protocolo, sino que son estándares de facto a nivel implementación.

Interrupt coalescing

También conocida como *Interrupt Moderation*, esta optimización ayuda a reducir el overhead de usar el mecanismo de interrupciones para señalizar el arribo de paquetes entrantes de la red. Para amortizar el costo de la interrupción, las NICs pueden acumular múltiples paquetes y notificar al procesador sólo una vez para un grupo de paquetes.

A velocidades de gigabit por segundo, la única forma de reducir significativamente la cantidad de interrupciones con esta técnica es incrementando el tamaño del buffer FIFO on-board para poder acumular más tramas (que ahora llegan con más frecuencia). Sin embargo, esto tiene un límite ya que incide directamente en la latencia que sufren los paquetes de red.

Por lo tanto, y debido a que demostró ser una forma efectiva de reducir el uso de CPU en el procesamiento de red, se ha evaluado la posibilidad de extender esta idea más allá del hardware. Por un lado, ya existen mecanismos híbridos que

combinan interrupciones con polling (ver capítulo 3) soportados por el kernel Linux 2.6. Por otro lado, se continúa experimentando con el concepto de llevar la idea de interrupt mitigation más allá en el software, por ejemplo procesando más de un paquete por invocación al código TCP/IP [8, 4].

Alternativas a las capas y reducción del costo del checksum

El modelo en capas ha sido considerado desde hace mucho tiempo no sólo como el marco conceptual básico en el diseño de protocolos, sino también como el modelo de ingeniería para la implementación. Sin embargo, varios trabajos de investigación han demostrado que no es una buena idea forzar la división en capas tanto en el diseño como en la implementación de una pila de protocolos de alta performance.

Ya es generalmente aceptado que una implementación de protocolos separada estrictamente en capas lleva a la ineficiencia porque previene el intercambio de información entre capas y de esta forma hace imposible que una capa optimice su comportamiento con respecto a las capas superiores e inferiores. Por ejemplo, considerando una capa de transporte que acepta un flujo de bytes de una aplicación, divide el flujo en paquetes y los envía a la red. Para optimizar la transferencia, la capa de transporte debería elegir el tamaño máximo de paquete posible. En particular, si la máquina destino se conecta directamente a través de una LAN, sólo un segmento de red estará involucrado en la transmisión y por lo tanto el emisor podría optimizar el tamaño de paquete para esa tecnología de red.

Se ha propuesto [26] que el diseño basado en capas no debe ser pensado como el diseño único y fundamental, sino como una de las aproximaciones a la ingeniería de protocolos, la cual debe ser comparada en función de la performance y simplicidad con respecto a otros métodos. Un ejemplo de esto es el modelo denominado *Integrated Layer Processing* (ILP) para integrar el procesamiento DTO (ver capítulo 4) de diferentes capas y formar un pipeline para mejorar la localidad de las referencias y reducir los accesos a memoria.

IPL desafía al esquema tradicional de capas de protocolos estructurados en una pila. Las pilas de protocolos en capas proveen una división funcional entre módulos de las distintas capas. Cada capa puede tener su propio almacenamiento privado de los datos y realizar manipulaciones de los datos de manera independiente, creando gran cantidad de tráfico a memoria innecesario.

IPL puede minimizar las referencias a memoria que resultan de operaciones DTO en diferentes capas, ya que las manipulaciones de diferentes protocolos son combinadas en un pipeline. Una palabra de datos es cargada en un registro de la CPU, luego es manipulada por múltiples etapas del procesamiento de datos mientras permanece en el registro, y finalmente es almacenada en memoria, todo esto antes de que la próxima palabra de datos sea procesada. De esta forma,

una serie de manipulaciones combinadas sólo referencia a memoria una vez, en lugar de acceder a memoria múltiples veces.

TCP/IP usa el checksum para asegurar que los datos del paquete no fueron corrompidos durante la transmisión. El cálculo del checksum en un procesador de propósito general se consideraba como una operación costosa (más tarde se relativizó esta afirmación), entonces se diseñaron formas de evitar este costo. El *Hardware Checksum Off-loading* (delegación del checksum al hardware) por un lado y la combinación de copia y checksum por otro, son dos técnicas que intentan minimizar los efectos de la operación de checksum necesaria sobre los datos de red, a la vez que representan ejemplos de aplicación de las ideas de IPL.

En el checksum Off-loading, la pila TCP/IP delega esta función a la NIC, si ésta lo soporta. Ya que implementar checksum en hardware es relativamente sencillo, ésto no agrega mucha complejidad o costo. Por lo tanto, muchas NICs modernas lo soportan, aunque se continúan evaluando sus beneficios reales. Por ejemplo, para paquetes extremadamente pequeños (de 64 bytes) un trabajo [4] demostró que la utilización de la CPU es en realidad mayor cuando se usa checksum por hardware (aunque también se logra mayor rendimiento). La reducción en la utilización de CPU esperada puede notarse mayormente con paquetes de mayor tamaño, ya que el checksum es una operación DTO.

La otra técnica utilizada [4] es que en lugar de ser implementadas como rutinas separadas, la operación de copia de los datos de red en memoria y el checksum puedan ser combinadas en una única operación. Ésto funciona leyendo cada palabra en un registro de la CPU, sumándola a un registro de suma y después escribiendo la palabra en su destino. Esta aproximación ahorra una operación de lectura y es una optimización bien conocida propuesta por primera vez por Van Jacobson. Básicamente, realizar el checksum cuando los datos son copiados en memoria incurre en un mínimo costo extra con respecto a la implementación del checksum por hardware.

Sin embargo, la implementación correcta de checksum y copia integrada (csum-copy, como se le llama a esta operación en el kernel Linux) no es trivial. En la recepción, puede haber dos situaciones potenciales: la aplicación invoca a la función read() antes de que lleguen datos (se pre-designan los buffers de la aplicación que almacenarán los datos), o los datos arriban antes de que la aplicación invoque a read(). Ésto implica dos caminos muy diferentes a seguir por los datos: En el caso donde la aplicación no ha designado los buffers (no se invocó a read()), el paquete es primero comprobado por checksum y luego copiado cuando los buffers de la aplicación están disponibles. En el otro camino de ejecución, el paquete es copiado y comprobado por checksum concurrentemente.

Además para que csum-copy suceda, deben cumplirse tres condiciones: los

números de secuencia deben estar en orden, el proceso *current*[1] debe ser el lector del socket y el buffer de esta aplicación lo suficientemente grande como para recibir los datos disponibles en el socket. Las primeras dos condiciones normalmente son verdaderas en transferencias de flujos de datos.

El camino de transmisión es más directo: csum-copy siempre se aplica en transferencias de flujos de datos. Se debe notar que el driver mantiene un registro de estado de la NIC mapeado en memoria que permite que TCP conozca el estado de los recursos e interrumpa la transmisión si es necesario. De esta forma el driver nunca desechará un paquete por falta de recursos en una transmisión y, por lo tanto, nunca se realizará una operación de checksum sobre un paquete que luego será descartado.

Large Segment Offload (LSO) o Transmit Side Offload (TSO)

La segmentación de buffers grandes de datos en segmentos más pequeños y el cómputo de la información de header TCP e IP para cada segmento es costoso si se hace en software. La mayoría de las NICs modernas soportan esta característica porque hacerlo en hardware no agrega demasiada complejidad. LSO es un estándar de facto en la versión 2.6 del kernel Linux y provee una reducción significativa en la utilización de la CPU.

Esta funcionalidad beneficia sólo al procesamiento de transmisión y sólo se usa cuando la aplicación envía datos más grandes que el Maximum Segment Size (MSS) que fue negociado por ambos extremos de la conexión durante la fase de establecimiento TCP.

Jumbo Frames

El MTU original de Ethernet se eligió por las altas tasas de errores y las bajas velocidades de transmisión de los medios físicos de hace más de dos décadas. De esta forma, si se recibe un paquete con errores, sólo debe reenviarse (como máximo) 1518 bytes. Sin embargo, cada trama requiere que el hardware y el software la procese. El uso de tramas más largas generalmente conlleva a un mayor rendimiento y menor uso de la CPU, al amortizar el NDTO (ver capítulo 4). Una aproximación para mejorar la performance de E/S ha sido a través de la introducción de Jumbo Frames.

Las Jumbo Frames permiten que el hardware Ethernet envíe y reciba tramas más grandes que 1518 bytes, generalmente de 9000 bytes. Esto reduce el número de paquetes que un sistema tiene que procesar y transferir a través del bus. Mientras que las Jumbo Frames son universalmente soportadas en todos los sistemas operativos, no han sido adoptadas fuera del entorno del data center. La mayor parte del debate sobre las Jumbo Frames se ha enfocado en la

[1]En Linux, se le llama proceso current al proceso que está ejecutándose actualmente

performance de LANs y el impacto del tamaño de las tramas en los requerimientos de procesamiento, performance del bus de E/S, memoria, etc. Sin embargo, diversos trabajos han planteado la importancia (principalmente para Grid y High Performance Computing) del tamaño de trama en la performance de redes WANs[2].

Zero-copy

Sabiendo que los accesos a memoria son operaciones costosas en términos de ciclos de CPU, existen diversos intentos por lograr operaciones de red sin copias en memoria. Como se mencionó anteriormente, estas técnicas se conocen como operaciones zero-copy.

Los datos a ser transmitidos no necesitan ser leídos por la CPU en todas las situaciones, permitiendo operaciones zero-copy. Linux implementa una versión simple de transmisiones zero-copy vía la system call sendfile(). Esta rutina hace uso del hecho de que los datos necesitados pueden residir en la Page Cache del VFS [7]. Una página de la Page Cache entonces funcionará como el socket buffer (estudiado en el capítulo 3) y los datos serán transmitidos directamente desde estos buffers a la NIC, usando DMA. Es decir, con la system call sendfile(), el kernel no necesita copiar los datos de usuario en los buffers de red. Esto evita la polución de cache de CPU con datos que probablemente no sean usados nuevamente, y disminuye la cantidad de ciclos necesarios para enviar un paquete.

Sin la ayuda del socket buffer como intermediario, la aplicación debe monitorear cuidadosamente el uso y liberación de las páginas de la page cache. Una página de datos enviados debe ser mantenida intacta hasta que las confirmaciones de TCP para los datos asociados hayan arribado. El servidor web Apache 2.0 provee soporte para el uso de sendfile() cuando el sistema operativo lo implementa.

Más allá de sus limitaciones, sendfile() es una de las formas mas sencillas y prácticas de lograr transmisiones zero-copy. Sin embargo no existe una solución similar para el camino de la recepción. Por ahora, la técnica TCP Offload Engine, explicada más adelante en este capítulo, indirectamente logra zero copy ya que todo movimiento de datos está localizado en la NIC. Los datos son finalmente copiados con DMA al buffer de la aplicación cuando la operación está completa. Por otro lado, un esfuerzo llevado adelante por la IETF busca colocar los datos en el sitio remoto directamente en los buffers de la aplicación, logrando de esta forma operaciones zero copy. Esta técnica se conoce como Remote Direct Memory Access y también se estudia más adelante este capítulo.

[2] Puede leerse una introducción a este tema en http://sd.wareonearth.com/p̃hil/jumbo.html

7.3 Propuestas más recientes

Las técnicas mencionadas ayudaron a mejorar la eficiencia en el procesamiento TCP/IP. Sin embargo, a velocidades de 10 Gbps, los problemas de overhead se agudizan, y se hace evidente la necesidad de nuevas técnicas.

Como ocurrió en escenarios previos del tipo *redes rápidas, máquinas lentas* (como la transición a 100 Mbps y a 1 Gbps), la necesidad de mejorar la performance ha disparado un interés renovado por parte de la industria en desarrollar asistencia por hardware para la NIC, incluyendo soluciones *stateless* y *stateful* para TCP, así como el soporte necesario del sistema operativo para soportar este hardware. Hasta ahora, la aceptación de la asistencia TCP stateful ha variado en el tiempo.

Las implementaciones stateful (como TCP Offload Engines, explicado más adelante) nunca lograron un lugar importante en el mercado o soporte generalizado en los sistemas operativos. Las razones principales incluyen costos, complejidad de la implementación, preocupaciones de seguridad, etc. Por otro lado, la asistencia stateless, incluyendo checksum offload y TSO, han logrado soporte universal en todos los sistemas operativos principales y se han convertido en un estándar de facto en las NICs para servidores de alta performance.

Estas tecnologías clave permitieron que el envío de paquetes sea mucho menos intensivo en uso de CPU que la recepción de paquetes. Lamentablemente, ninguna de estas optimizaciones pueden ser aplicadas para mejorar la performance en la recepción. Esto no es de sorprender, ya que la delegación de funcionalidades de recepción en el hardware son menos sencillas de implementar debido a potenciales recepciones fuera de orden, entre otras razones. La utilización de la CPU en recepción, como consecuencia, se convierte en el cuello de botella que impide que las NICs de 10 Gbps alcancen las velocidades del estándar, especialmente con una MTU de 1500 bytes. Sin embargo están surgiendo NICs Ethernet de 10 Gbps (Intel y Neterion, por ejemplo) que poseen soporte por hardware de algunas operaciones de recepción.

A continuación se describen diversas técnicas modernas, algunas de ellas contradictorias entre sí, que intentan acercar las posibilidades de procesamiento de red del hardware y el software a las capacidades de transmisión de las redes de alta velocidad actuales.

7.3.1 Mejorando la recepción en el driver/NIC

Actualmente se intenta aliviar el overhead adicional del procesamiento de recepción de paquetes con la suma de tres características adicionales, cada una destinada a resolver un cuello de botella particular en el procesamiento de recepción, mejorando el rendimiento y/o disminuyendo la utilización de CPU. Afortunadamente, pueden ser implementadas sin cambios radicales a la forma

en que la pila de Linux funciona actualmente.

Headers Separados

Para transmisiones sobre la red, varias capas de headers son agregadas a los datos de la aplicaciones. Un ejemplo común consiste de un header TCP, un header IP y un header Ethernet. Cuando se recibe un paquete, como mínimo el controlador de red debe examinar el header Ethernet y el resto del paquete puede ser tratado como datos opacos. Por lo tanto, cuando el controlador DMA transfiere un paquete recibido, típicamente copia los headers TCP/IP y Ethernet junto con los datos de la aplicación, todo a un único buffer.

El reconocimiento de headers de protocolos de alto nivel puede permitir al controlador realizar ciertas optimizaciones. Por ejemplo, todos los controladores modernos reconocen headers IP y TCP, y usan este conocimiento para realizar validaciones de checksum por hardware. Usando esta característica, el controlador puede particionar el paquete entre los headers y los datos, y copiarlos en buffers separados en memoria principal. Esto tiene varias ventajas. Primero, permite que tanto los headers como los datos sean alineados en memoria de manera óptima. Segundo, permite que el buffer para el paquete de red se construya con un pequeño buffer para el header alocado por slab cache (ver capítulo 3), más un buffer de datos más grande alocado por página. Realizar estas dos alocaciones es más rápido que hacer una alocación grande por slab cache, debido a la estrategia subyacente de alocación de memoria, conocida en el kernel Linux como Buddy Alocator[7].

Receive-side scaling

Esta técnica permite que los sistemas multiprocesadores procesen más eficientemente el tráfico de red en sus CPUs. Se implementa con múltiples colas de recepción en la NIC que son asociadas a los diferentes procesadores en el sistema. La cola para un paquete dado es elegida computando una función hash de ciertos campos en los headers de los protocolos. Esto resulta en que todos los paquetes para un única conexión TCP sean colocados en la misma cola.

La próxima generación de NICs de Intel tendrán múltiples colas de recepción ya que esto mejora la escalabilidad a través del paralelismo a nivel de conexión y una mayor afinidad de conexiones a procesadores.

Large Receive Offload

Esta característica implementada por hardware en la NIC asiste al host en procesar paquetes entrantes agrupándolos. Es decir, el host recibe la misma cantidad de datos, pero en porciones más grandes, de esta forma la CPU debe procesar

una menor cantidad de paquetes pero de mayor tamaño y, por lo tanto, se reduce el uso de CPU por costos del tipo NDTO.

7.3.2 TOE

La iniciativa denominada TCP Offload Engine [29, 14], pero más conocida como TOE, propone el uso de hardware especializado en la propia placa de red para realizar algunas o todas las tareas asociadas con el procesamiento de los protocolos y de esta forma aliviar al procesador principal del sistema.

Desde hace tiempo, los fabricantes han argumentado que TOE incrementa el rendimiento de red del servidor mientras que reduce la utilización de CPU, sin embargo, esta afirmación es relativa. Para algunos escenarios, especialmente transferencias que involucran a pocas conexiones y tamaños de paquetes grandes usar TOE puede mejorar la eficiencia.

Si bien varios modelos de placas TOE han aparecido en el mercado, en general la idea no resultó un éxito comercial. El hecho de implementar los protocolos en la propia placa impide modificarlos fácilmente, con lo cual se dificultan posibles optimizaciones por cuestiones de performance o seguridad. Además el hardware con el que se implementan los procesadores de propósito general tiende a incrementar la relación funcionalidad/precio a un ritmo más favorable que el del hardware específico colocado en los dispositivos TOE. La ley de Moore se aplica a procesadores de propósito general y de gran volumen, y el hardware TOE no pertenece a ese conjunto.

Por otro lado, como los dispositivos TOE se integran al subsistema de I/O de la plataforma del servidor, la latencia a la memoria principal es relativamente alta comparada con la latencia de la CPU principal. Por lo tanto, las soluciones TOE se han confinado a situaciones en las cuales grandes bloques de datos se envían en paquetes de 8 KB o más. Estos payloads grandes requieren menos interacciones con la plataforma y, por lo tanto, el TOE sufre menos overhead por latencia de memoria. Las aplicaciones de almacenamiento y bases de datos corporativas generalmente utilizan tamaños de paquetes relativamente grandes. Sin embargo, la tendencia clara en los data centers actuales es en dirección a paquetes de red con payloads pequeños. Web services, pasaje de mensajes, comunicación en clusters, RFID, y mensajes en tiempo real, todos usan paquetes más pequeños, típicamente menos de 1 KB. Este tráfico no es bien manejado por soluciones TOEs [14, 15, 23].

RDMA

Remote Direct Memory Access [28, 14] (RDMA) es una tecnología que permite que el sistema que envía los datos los ubique en una posición específica en la memoria del sistema que los recibe. De esta forma, se requiere menos interven-

ción de la CPU del receptor para mover los datos entre buffers de su memoria. Según las críticas [23], RDMA muestra potencial pero necesita resolver algunas preocupaciones de la comunidad antes de que sea ampliamente adoptada:

- Es una pila de protocolos de alto nivel que requiere una capa de transporte confiable como TCP. Por esta razón, RDMA se asocia con TOE.
- RDMA es un protocolo punto a punto. Se necesita instalar NICs especializadas en cada servidor del data center.
- RDMA necesita que las aplicaciones sean portadas a su API para que puedan aprovechar sus beneficios.
- Finalmente, existe cierta preocupación sobre el riesgo de seguridad que implica la habilidad de un nodo para colocar datos en la memoria de otro nodo en la red.

7.3.3 Onloading

La alternativa al offloading es mejorar la habilidad inherente a la arquitectura del servidor para procesar paquetes TCP/IP eficientemente y a velocidades muy altas. Esta alternativa se conoce como Onloading.

El Onloading propone al procesador del sistema como el principal recurso para el manejo de tráfico de red y no sugiere grandes cambios en el hardware o en las aplicaciones. Los procesadores de propósito general, con sus economías de escala, tienen la ventaja de ser flexibles, extensibles y escalables. Su programabilidad y la disponibilidad de un amplio conjunto de herramientas de programación hace posible agregar nuevas características, protocolos y aplicaciones, permitiendo que el software de red se adapte fácilmente a estándars cambiantes, nuevas optimizaciones y modificaciones en el diseño.

A continuación se explica una implementación del concepto de Onloading que puede tomarse como la referencia más importante: Intel I/O Acceleration Technology [14, 15].

Intel I/O Acceleration Technology

Este proyecto implica cinco vectores principales de investigación que ya produjeron resultados positivos en reducir el requerimiento de procesamiento para redes de alto rendimiento. Con respecto al overhead del sistema operativo, Intel está evaluando la posibilidad de dedicar CPUs exclusivamente al procesamiento de red y buscando la forma de optimizar la pila TCP/IP. Para superar el problema de memoria, se desarrolló una pila de referencia especial para el procesamiento TCP/IP conocida como Memory-Aware Reference Stack (MARS). El desarrollo de MARS introduce mecanismos innovadores para traer los datos

más cerca de la CPU antes que sean accedidos o para superponer computación útil al tiempo de latencia del acceso a memoria. Además de los algoritmos de *prefetching* ya disponibles en las CPUs actuales, MARS aprovecha otras técnicas de reducción de latencia de memoria, como se estudia más adelante.

Procesamiento de la pila de protocolos

El proyecto *Embedded Transport Acceleration* (ETA) en Intel Labs ha desarrollado una arquitectura prototipo en la cual todo el procesamiento de paquetes de red se hace en uno o mas procesadores dedicados. La arquitectura expone una interfaz de comunicación directamente a los procesos de usuario[3], reduciendo las copias de datos y evitando las capas del sistema operativo para la mayoría de las operaciones de E/S.

Optimización de la pila de protocolos

Intel ha estado trabajando en optimizar el camino de procesamiento que suelen seguir los paquetes a través de la pila de protocolos. Se están re-implementando algunas funciones para adaptarlas a las nuevas posibilidades que ofrecen los procesadores modernos en arquitecturas paralelas, y se están rediseñando las estructuras para que logren mayor eficiencia de memoria cache, todo esto utilizando técnicas modernas de codificación y compiladores de última generación. El resultado será, según Intel, una pila optimizada que reduce significativamente la cantidad de ciclos de CPUs necesarios para procesar un paquete.

Threads Livianos

Para tolerar los eventos de mucha latencia como accesos a memoria y copias de datos, MARS utiliza threads de granularidad muy fina (strands, según terminología de Intel) que se ejecutan en el contexto de un único thread del sistema operativo. Los strands pueden procesar paquetes independientes (pertenecientes a conexiones diferentes) o realizar operaciones de bookkeeping.

Cuando un strand incurre en un evento de gran latencia, como un fallo de cache, se cambia a otro strand para superponer la latencia de memoria con computación útil. Para que esto sea efectivo, el overhead del switching de strands debe ser extremadamente pequeño (una fracción del tiempo de acceso a memoria).

Es decir, en lugar de proveer múltiples contextos de hardware (CPUs virtuales para el sistema operativo), como la tecnología Hyper-Threading, un único contexto de hardware contiene a la pila de red con múltiples threads controlados

[3]Similar a Virtual Interface Architecture (http://www.intel.com/intelpress/chapter-via.pdf)

por software. En términos de implementación, la detección del acceso a memoria (un fallo de cache u operación de copia) y el switching a otro thread puede ser implícito (requiriendo soporte de hardware) o explícito (implementado en software).

Acceso directo a la cache

DCA (Direct cache access) promueve el *modelo Push*[4] para las transferencias de datos entre la NIC y la CPU. Las plataformas convencionales colocan los descriptores, headers y datos de red entrantes en memoria antes de notificar a la CPU que los datos han arribado. Como resultado, cuando la CPU accede a estos datos, sufre de fallos de cache y las latencias asociadas a las lecturas de memoria.

DCA ayuda a reducir los ciclos perdidos por latencia de acceso a memoria y reducir el uso del ancho de banda a memoria haciendo que los datos se coloquen directamente en la cache de la CPU. Los accesos a memoria se logran reducir hasta un 40 % [14, 15].

Copias Asincrónicas en Memoria

Esto se logra con un componente de hardware que permite que las copias sucedan asincrónicamente con respecto a la CPU. Una vez que la operación de copia es planificada en este dispositivo la CPU puede utilizar otro strand para procesar otro paquete. Las consideraciones de eficiencia para la implementación de un dispositivo de copias incluyen los costos para iniciar y notificar la terminación de una copia. Por lo tanto, se buscan mecanismos de bajo overhead para realizar estas funciones, como por ejemplo integrar el hardware de copias en los procesadores.

I/OAT de intel delega las copias de datos a un componente DMA, un dispositivo dedicado a hacer copias en memoria. Mientras que el componente DMA realiza la copia de datos de una ubicación de memoria a otra, la CPU esta libre para proceder con el procesamiento del próximo paquete u otra tarea pendiente. El componente DMA se implementa como un dispositivo PCI, incluido en el chipset, y tiene múltiples canales DMA independientes con acceso directo a la memoria principal. Cuando se completa una copia puede generar opcionalmente una interrupción. El soporte en el kernel Linux implementa una interfaz genérica de copias asincrónicas en memoria que podría ser usada por otras partes del kernel, además del subsistema de red.

Usando este mecanismo, el procesamiento de paquetes por la CPU y la copia de datos por el componente DMA se realizan en paralelo. Sin embargo, para que exista un buffer de usuario disponible para que comience la copia asincrónica

[4] http://www.usenix.org/events/sruti05/tech/full_papers/duan/duan_html/node4.html

temprana, el proceso de usuario debe invocar a read() para designar un buffer de destino. Si el proceso utiliza las llamadas select() o poll() para esperar datos, los buffers de usuario no están disponibles hasta que los datos hayan arribado y esto reduce el paralelismo logrado. El trabajo actual y futuro relacionado con APIs de red asincrónicas permitirán un mejor aprovechamiento de las copias asincrónicas en memoria.

7.4 Arquitectura de los servicios y la E/S asincrónica

7.4.1 Modelos de aplicaciones

Las aproximaciones actuales de implementación de aplicaciones para redes de alta performance requieren técnicas especiales para manejar los altos niveles de concurrencia. A continuación se mencionan los pasos lógicos realizados por un servidor web para manejar un único requerimiento de un cliente. La mayoría de este tipo de aplicaciones siguen pasos similares. Para simplificar el esquema se asume una conexión persistente iniciada por el cliente.

1. Esperar y aceptar una petición de conexión.
2. Leer un requerimiento entrante de la red. Si el cliente no tiene más requerimientos, cierra su extremo de la conexión, la operación read() que el servidor invocó sobre el socket que representa la conexión retorna EOF (End Of File) y se cierra la conexión.
3. Parsear el requerimiento.
4. Para requerimientos estáticos, chequear la page cache y, eventualmente, abrir y leer el archivo.
5. Para requerimientos dinámicos, computar el resultado.
6. Enviar la respuesta al cliente.
7. Volver al paso 2.

En entornos de gran escala, un servidor debe manejar varios miles o decenas de miles de conexiones simultáneas, multiplexar rápidamente conexiones que están listas para ser servidas, y despachar E/S de red a muy alta velocidad. Debe tenerse en cuenta que varios de los pasos listados anteriormente pueden bloquear al proceso por que requieren interacción con el host remoto, la red, una base de datos, el disco, o cualquier otro subsistema.

Una aproximación posible al problema de la concurrencia es multiplexar conexiones simultáneas usando procesos o threads que son bloqueados por el

sistema operativo cuando una system call se bloquea. La multiplexación de conexiones ocurre cuando el sistema operativo cambia de contexto a un thread que está listo para ejecutarse porque ha arribado un evento que el thread estaba esperando bloqueado. Esto es conocido como un modelo MP (multi-process) o MT (multi-thread) [21].

Una solución alternativa para satisfacer este requerimiento es usar una arquitectura Single Process Event Driven [21] (SPED). Una arquitectura SPED usa un único proceso y sólo realiza system calls que no se bloquean. Esto permite que el proceso realice múltiples conexiones concurrentemente sin ser bloqueado en una operación de E/S. Un modelo de programación de E/S no bloqueante requiere un mecanismo de notificación de eventos como select(), poll() o epoll() [7] (la última sólo en Linux) para determinar cuándo puede realizarse una system call sin bloquearse. En entornos multiprocesadores pueden usarse múltiples copias del proceso servidor SPED para obtener una buena performance en relación a otras arquitecturas.

Un claro ejemplo de la evolución en el diseño de los servidores es Flash [21]. El servidor web Flash implementa una arquitectura conocida como Asymmetric Multi-Process Event Driven (AMPED). Esta es una arquitectura híbrida que combina E/S de sockets manejada por eventos (SPED) con procesos auxiliares dedicados a realizar los accesos a disco (bloqueantes) en nombre del proceso principal, y que fue diseñada específicamente para máquinas SMP.

7.4.2 AIO

Para lograr una concurrencia escalable sin el overhead de los threads por software, se están proponiendo APIs que poseen una semántica de E/S asincrónica (Asynchronous I/O o AIO) para operaciones de sockets y archivos. Las aplicaciones realizan operaciones sin ser bloqueadas y sin verificar previamente el estado del descriptor de socket o archivo, luego reciben eventos asincrónicos que señalizan la terminación de una operación de E/S previamente invocada. Este modelo permite que múltiples operaciones E/S estén realizándose concurrentemente sin que el proceso sea bloqueado por el sistema operativo.

La E/S asincrónica divide cada operación de E/S en dos fases distintas: generar un requerimiento de operación y recuperar la información del evento de terminación. Un thread de una aplicación puede ejecutar una llamada no bloqueante que genera un requerimiento, generalmente con un mensaje en una cola de trabajos, para que se realice una operación de E/S. Cuando la operación eventualmente se completa, un evento de terminación es despachado asincrónicamente a la aplicación, típicamente a través de una cola de eventos, para indicar el resultado de la operación. El procesamiento de E/S con fases separadas permite que un thread realice otro procesamiento entre el momento en que invoca una operación y el momento en que recibe el evento de terminación. El otro

procesamiento puede incluir iniciar más operaciones y procesar otros eventos de terminación para diferentes operaciones de E/S, incrementando de esta forma el nivel de concurrencia de E/S con respecto a la E/S sincrónica o bloqueante.

A diferencia de la E/S sincrónica, AIO no bloquea al thread de la aplicación que la invoca, pero también se debe distinguir AIO de la E/S no bloqueante. En esta última la aplicación se asegura que la operación E/S no se bloqueará verificando antes el estatus del archivo o socket. Por ejemplo, en un sistema Unix, un socket o un archivo pueden ser configurados en modo no bloqueante y las llamadas a read() o write() que no puedan realizarse sin bloquearse retornarán inmediatamente (con el valor de retorno EWOULDBLOCK). Las aplicaciones también pueden invocar a select() o poll() para detectar qué llamadas de read() o write() pueden ser realizadas sin bloquear al proceso que las invoca. Aunque usando estos métodos no bloqueen a los procesos, estas llamadas pueden retornar sólo resultados parciales, por ejemplo porque se alcanzó el límite del buffer del socket, en cuyo caso las operaciones necesitan ser invocadas múltiples veces. En contraste a la E/S no bloqueante, con operaciones AIO las aplicaciones no necesitan verificar previamente los descriptores o manejar resultados parciales.

Actualmente, existen varias implementaciones de APIs asincrónicas de E/S como por ejemplo Windows asynchronous I/O de Microsoft, POSIX AIO y Linux AIO. Estas APIs están pensadas para una pila de E/S (sockets y archivos) basadas en el kernel. Recientemente el Open Group[5] ha comenzado a definir las extensiones asincrónicas para la API de sockets para Unix, diseñada para implementaciones donde las operaciones de E/S no se implementan en el kernel (por ejemplo, usando adaptadores inteligentes de Infiniband). Las implementaciones en espacio de usuario buscan, entre otras cosas, reducir las copias en memoria y los overhead introducidos por el sistema operativo.

Como se dijo anteriormente, hay dos aproximaciones básicas a estructurar aplicaciones para que escalen a operaciones altamente concurrentes: threads y eventos. Aunque se puede lograr una buena performance con threads, no existe una implementación de threading de propósito general que escale eficientemente a un número realmente grande de threads [25]. En la aproximación alternativa basada en eventos, el sistema operativo o dispositivo de E/S despacha eventos a las aplicaciones para indicar cambios en el estatus del sistema. Las aplicaciones se estructuran como máquinas de estado en las cuales una transición ocurre cuando la aplicación procesa un evento. Al exponer colas de trabajos y eventos a las aplicaciones, el modelo de programación naturalmente permite que las aplicaciones procesen eventos y realicen operaciones usando políticas de scheduling específicas para mejorar el manejo de los recursos y la performance, en función de los objetivos.

[5] www.opengroup.org

Capítulo 8

Un prototipo básico y los resultados preliminares

8.1 Introducción

En el capítulo 6 se observó que la intrusión del sistema operativo puede consumir gran parte de la capacidad de cómputo de una computadora y se llegó a la conclusión de que los mecanismos tradicionales que permiten implementar la protección y virtualización en el kernel del sistema operativo deben ser repensados. Dos de estos mecanismos son las interrupciones del hardware y las system calls.

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 propone dedicar uno o varios CPUs (o cores) a realizar exclusivamente actividades del kernel, de manera tal de no depender de mecanismos costosos para la invocación de sus funciones. Este capítulo presenta una modificación al subsistema de red del kernel Linux 2.6 basada en esta idea. Además se exponen las mediciones realizadas comparando distintas configuraciones del kernel estándar y de un kernel modificado.

8.2 Diseño e implementación del prototipo

A continuación se documenta un primer prototipo que implementa una de las ideas que se han estudiado. Lejos de poseer un diseño elaborado y una implementación optimizada, el prototipo presentado a continuación se ofrece sólo como

una prueba de concepto. Sin embargo, permitió realizar ciertas mediciones comparativas de utilidad y concluir en que la idea de un kernel activo puede reducir los factores que impiden la escalabilidad.

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 poco overhead y no intrusivo, como la memoria compartida.

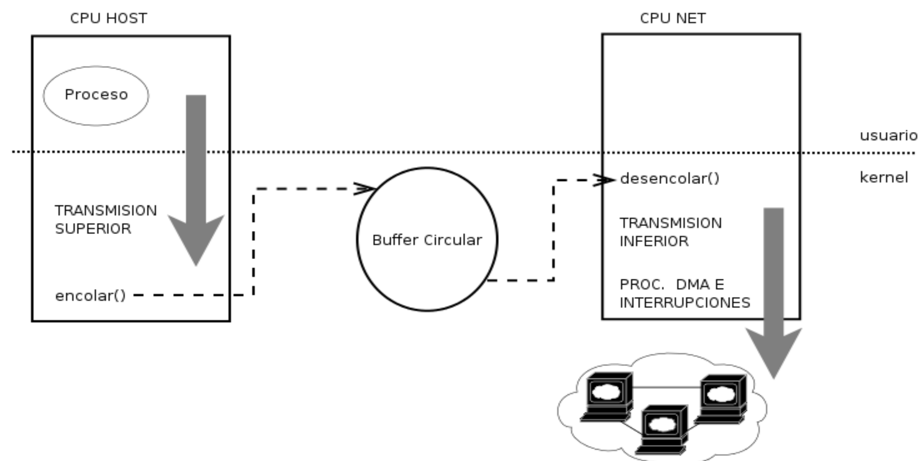

Figura 8.1: Diagrama del prototipo - Proceso de transmisión

Una CPU-NET no necesita ser multiplexada entre otras aplicaciones y/o threads del sistema operativo, con lo cual puede ser optimizada para lograr la mayor eficiencia posible en su función. Por ejemplo, es posible evitar el overhead 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 (como se explicó en el capítulo 5), sino que se ejecuta sincrónicamente en un subconjunto reducido de CPUs (posiblemente sólo una CPU, dependiendo del tráfico a procesar).

De esta forma se incrementa la eficiencia en el procesamiento de los protocolos de red, ya que se reduce la intrusión por mecanismo del sistema operativo. Las aplicaciones ya no serán interrumpidas varias miles de veces por segundo[1]

[1] A una velocidad de 10Gbps, con tramas de 1500 bytes (12000 bits), se reciben 833333 tramas por segundo. Esto significa casi 1 millón de interrupciones de recepción de red por

y el scheduler no sufrirá las interferencias causadas por los altos requerimientos de procesamiento que implica el tráfico de alta performance.

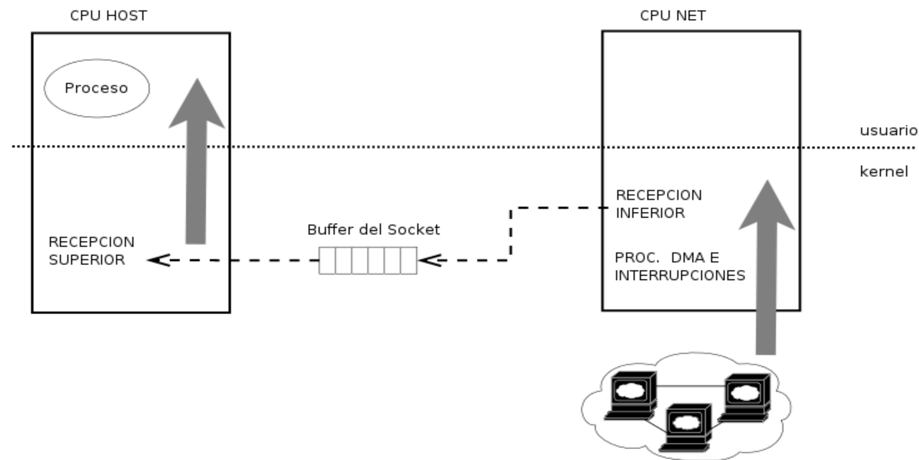

Figura 8.2: Diagrama del prototipo - Proceso de recepción

No se reinventaron los algoritmos de red, sólo se reusaron las funciones que ofrece el kernel para el procesamiento de red, en otros contextos de ejecución. 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 aisla una CPU para ejecutar este código de manera sincrónica y controlada. Con esto se logra reducir el tiempo consumido por las funciones del kernel que no realizan trabajo útil para el usuario, como pueden ser las funciones del subsistema de interrupciones, las que planifican y disparan las softirq, y las que realizan los cambios de contexto, entre otras.

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 system call recv() que realiza el proceso servidor que se ejecuta en la CPU-HOST. Por lo tanto, siempre se realiza el procesamiento de recepción inferior (tal cual se estudió en el capítulo 3) y el procesamiento de las interrupciones en la CPU-NET.

Con respecto a la transmisión, se implementaron dos variantes, dependiendo de cuánto procesamiento 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 cons-

segundo, si no se utiliza algún mecanismo de reducción de interrupciones.

tantemente 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.

Se modificó el kernel Linux 2.6.17.8 para implementar la arquitectura separada de CPU-HOST y CPU-NET en un servidor SMP de dos CPUs AMD Opteron. En particular, fue necesario agregar código en los siguientes archivos

- net/core/dev.c
- include/linux/interrupt.h
- net/ipv4/tcp_output.c
- drivers/net/tg3.c, drivers/net/forcedeth.c, drivers/net/r8169.c

Se creó un módulo del kernel que dispara el thread que toma el control de la CPU1, la cual cumplirá el rol de CPU-NET. Las interrupciones de las placas de red se desactivan y se verifican por polling, y el resto de las interrupciones son ruteadas a la CPU-HOST usando los mecanismos de enrutamiento ofrecidos por el IO/APIC externo. De esta forma, la CPU-NET no debe manejar eventos asincrónicos no relacionados con la red. Sin embargo, la interrupción del timer local APIC 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 conjunto de CPUs-NET.

8.3 Algunas mediciones y resultados preliminares

8.3.1 Máquinas de prueba

Las pruebas se realizaron sobre máquinas Dual CPU, con 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 (128 bits de datos y 16 bits de ECC). 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.

Con respecto a la entrada/salida, la motherboard ofrece cuatro slots PCI v2.3 de 32 bits a 33Mhz y dos slots PCI-E (uno 4x y otro 16x) conectados directamente al chipset integrado (un único chip) nForce 2200 de Nvidia. También

se incluyen dos NICs gigabit con soporte para full duplex aunque, sorprendentemente, son dos controladores Ethernet de fabricantes diferentes y están integradas de maneras distintas. Un controlador es el modelo BCM5705 de Broadcom y comparte el bus PCI al cual se conectan los slot mencionados anteriormente, como se puede apreciar en la figura 8.3, presentada más adelante. El otro controlador es el 88E1111 de Marvell y goza de una conexión exclusiva al chiset de Nvidia. Finalmente se utilizó una tercer NIC con un chip Realtek modelo 8169 de Gigabit y con una interfaz PCI. Como se explica posteriormente, el hecho de que las placas de Broadcom y Realtek compartan el mismo bus PCI tiene como consecuencia directa un bajo rendimiento de red logrado por estas placas.

Además de la máquina descripta, que funcionó como servidor de todas la pruebas, se usaron máquinas similares como clientes, todas conectadas a través de un switch 3COM con 8 bocas full duplex de gigabit.

8.3.2 Herramientas de Medición e Instrumentación: Sar, Oprofile y Netperf

Para realizar las mediciones del rendimiento de red, el costo de procesamiento y el profiling del kernel se utilizaron las siguientes 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 durante un tiempo determinado gran cantidad de indicadores de performance en una máquina SMP identificándolos por CPU, y luego ofrece una media de esos datos.
- Oprofile [2] 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 vez que se obtiene una cantidad configurable de muestras obtenidas de un evento dado.

8.3.3 Las pruebas realizadas y algunos comentarios sobre los primeros resultados

Es importante aclarar que para las pruebas no se optimizó la configuración del kernel para maximizar el rendimiento de los enlaces. No es una tarea trivial lograr velocidades máximas sostenidas cercanas a 1 Gbps con las configuraciones

por defecto del kernel y/o de las aplicaciones de red utilizadas, usando TCP/IP. Cuestiones tales como tamaños de las ventanas TCP, tamaño de los buffers de los sockets, y algunas otras opciones (generalmente configurables desde el sistema de archivos virtual /proc) permiten elevar el rendimiento máximo logrado, sin embargo, no es el objetivo de este trabajo optimizar los enlaces, sino medir la eficiencia con respecto al procesamiento. Es decir, se busca reducir la relación GHz / Gbps en lugar de maximizar el rendimiento de red. Por lo tanto, se podrá observar que los enlaces llegan a mantener entre 700 y 750 mbps en las pruebas donde se usan dos NICs.

Para las pruebas A, B y C se utilizan 2 NICs (Eth0 es la NIC Realtek y Eth2 es la NIC Nvidia) y el kernel 2.6.17.8 estándar, sin modificaciones. Se transmiten datos durante dos minutos y luego se calculan valores promedios.

Prueba A) Se activa sólo una CPU de la máquina SMP para todo el procesamiento. Esto se implementa combinando dos configuraciones:

- Por un lado, se envían dos parámetros al kernel en el momento de la inicialización. *isolcpus=0* le indica al scheduler general del kernel SMP que sólo utilice la primer CPU para sus funciones y *noirqbalance* desactiva las funciones de balanceo de interrupciones del I/O APIC.
- Por otro lado, se rutean todas las interrupciones generadas por el hardware a la primer CPU. Esto se logra configurando la tabla de enrutamiento de interrupciones del I/O APIC para que todas las interrupciones sean redistribuidas al APIC local de la CPU0.

De esta forma todo el procesamiento de red, tanto procesos de usuario como threads y softirqs del kernel, se ejecutan en la primer CPU de la máquina y la otra CPU permanece ociosa todo el tiempo. La máquina nunca muestra más de un 50 % de actividad de sus procesadores.

Para las próximas dos pruebas, se activan las dos CPUs para el procesamiento de red, aunque se distribuye la carga de diferentes formas para estudiar el rendimiento de red y costo de procesamiento de cada configuración.

Prueba B) En este caso se usan ambas CPUs para procesar la actividad del kernel y procesos de usuario. Las interrupciones se distribuyen entre los procesadores, y los procesos de usuario se migran cada vez que el scheduler decida balancear la carga. Ésta es la configuración por defecto de los kernels que acompañan a las distribuciones más utilizadas (como Fedora y Debian) del sistema operativo GNU/Linux y resulta ser la opción menos eficiente a la hora de procesar tráfico de red.

Prueba C) En esta prueba se asocia el procesamiento de cada NIC con una CPU en particular. Es decir, en una misma CPU se ejecutan: el manejador de interrupciones de una de las NICs, la softirq correspondiente y el proceso de Netperf que atiende el servidor asociado a la dirección IP de esa NIC. Básicamente

se logra una afinidad total (como se estudió en el capítulo 5) del procesamiento de red de una NIC a una CPU. Esto se implementó en dos pasos:

- La afinidad de procesos de usuario se logra sin modificar el código fuente de la herramienta Netperf. El kernel Linux ofrece system calls que permiten especificar una máscara que indica el subconjunto de CPUs válidas para la potencial migración de un proceso dado. Es posible usar un simple programa, cuyo código fuente se adjunta a este trabajo, que recibe el PID (Process ID) de un proceso y la máscara que se desea asociar a ese proceso, e invoca a la system call correspondiente.

- La afinidad de interrupciones se implementa configurando el I/O APIC. Después de identificar el número de irq correspondiente a cada NIC, se configura la tabla de enrutamiento de interrupciones del controlador para que distribuya las interrupciones a cada CPU en función de su origen, permitiendo que cada CPU atienda interrupciones de una NIC distinta. Del resto se encarga el mismo kernel ya que la ejecución de las softirqs se planifica en la misma CPU que recibió la interrupción del hardware (como se estudió en el capítulo 3).

En las pruebas D y E se utiliza el prototipo implementado. En ambos casos se dedica la CPU1 a realizar procesamiento de red de manera exclusiva, sin embargo las pruebas difieren en cuanto al momento, en el camino del procesamiento de transmisión, en el cual son interceptados los paquetes para continuar su procesamiento en la CPU1 (en este momento resulta útil recurrir a la división funcional del procesamiento TCP/IP que se planteó en el capítulo 3).

En ambas pruebas se continúan procesando las copias desde y hacia el espacio de usuario (es decir, el procesamiento superior) en la CPU-HOST. Como se mencionó previamente y como se confirma en el profiling mostrado en esta sección, la operación de copia desde y hacia el espacio de usuario es una de las más costosas, con lo cual el prototipo puede ser mejorado sustancialmente si se reducen o evitan las copias. En el capítulo 7 se presentan algunas ideas que intentan lidiar con este problema.

Prueba D) En la prueba D se utiliza la CPU-NET para realizar el procesamiento de los siguientes componentes de TCP/IP:

- Procesamiento de interrupciones

- Recepción Inferior

Prueba E) En la prueba E se utiliza la CPU-NET para realizar el procesamiento de los siguientes componentes de TCP/IP:

- Procesamiento de interrupciones

- Recepción Inferior
- Transmisión Inferior

Es decir, se realiza en la CPU-NET todo el procesamiento que no esté vinculado a las copias desde y hacia el espacio de usuario.

En el cuadro 8.1 se presenta una comparación entre los resultados de las cinco pruebas mencionadas anteriormente. El significado de cada línea, para cada prueba, es el siguiente:

- **Rendimiento** representa el rendimiento total promedio de red, sumando el rendimiento de las dos NICs, medido en Megabits por segundo.
- **%Idle** es el porcentaje promedio del poder de cómputo de la máquina que queda disponible.
- **Cswch/s** es la cantidad promedio de cambios de contexto por segundo.
- **Intr/s** es la cantidad promedio de interrupciones recibidas por segundo.

En las transcripciones de las pruebas, entregadas con este trabajo, se pueden encontrar resultados con otro nivel de detalle. Por ejemplo, algunos de estos indicadores están discriminados por CPU.

Cuadro 8.1: Comparación de pruebas A, B, C, D y E

medidas\pruebas	A	B	C	D	E
Rendimiento(Mbps)	1379	1504	1482	1490	1418
%Idle	50,03	15,94	42,14	22,61	25,19
Cswch/s	44617,77	50454,35	222893,68	131499,63	101608,68
Intr/s	121823	105444	133873	0	0

Se puede observar que mientras el rendimiento se mantiene casi constante, los otros valores llegan a variar significativamente en algunos casos. Para comprender los motivos de estos resultados, se presenta a continuación el resumen de los datos arrojados por el profiling del kernel durante las pruebas realizadas.

En todas las pruebas, el profiling general del sistema (que incluye datos sobre los procesos de usuario, librerías y módulos del kernel, además del binario del kernel mismo, y que se entrega junto con este trabajo) permitió comprobar que el kernel consume cerca del 70 % de la CPU. Por lo tanto, se hizo hincapié en el profiling del kernel. Es decir, en los próximos cuadros se muestra el costo de las funciones del archivo binario que implementa el kernel, pero no se incluyen los costos de los drivers de las NICs (por ejemplo, no se incluye el costo del

manejador de interrupciones de cada NIC) o de los procesos que implementan los servidores de Netperf.

Los cuadros de profiling se componen de cinco columnas. Cada línea muestra datos de un símbolo en particular del kernel y de izquierda a derecha, estas columnas tienen el siguiente significado:

- Cantidad de muestras, incrementadas cada 400000 eventos del tipo CPU_CLK_UNHALTED, es decir, ciclos ocupados en computación útil (y no esperando por datos de memoria). Esta columna permite estimar qué funciones consumieron mayor cantidad de tiempo de CPU.
- Porcentaje del valor anterior con respecto al total.
- Cantidad de muestras, incrementadas cada 10000 eventos del tipo DATA_CACHE_MISSES. Esta columna permite estimar qué funciones muestran una baja localidad de las referencias a memoria.
- Porcentaje del valor anterior con respecto al total.
- Nombre del símbolo. En este caso son funciones del kernel.

Cuadro 8.2: Profiling del kernel durante la prueba A

muestras	%	muestras	%	símbolo
69213	18.2416	58248	55.5786	__copy_to_user_ll
53019	13.9735	1942	1.8530	handle_IRQ_event
21261	5.6035	2779	2.6516	tcp_v4_rcv
12545	3.3063	1364	1.3015	__do_IRQ
11823	3.1160	460	0.4389	qdisc_restart
11375	2.9980	267	0.2548	_spin_lock
9692	2.5544	1514	1.4446	irq_entries_start
9555	2.5183	1709	1.6307	schedule
8774	2.3125	1168	1.1145	_spin_unlock
8430	2.2218	3066	2.9255	kfree
7924	2.0884	3322	3.1698	tcp_rcv_established

Como se observa, entre las funciones del kernel que consumen más CPU se encuentran aquellas relacionadas con las copias físicas en memoria (__copy_to_user_ll()), la administración de interrupciones (handle_IRQ_event(), __do_IRQ()) y los mecanismos de sincronización (_spin_lock()).

Cuando se comparan la prueba A con la prueba B, se observa un incremento significativo del uso de CPU (se reduce %Idle del 50 % al 15 %) que no

es proporcional con el incremento del rendimiento de red. Al estudiar cómo se genera este aumento en el uso de CPU se observa un factor clave: la función spin_lock_irqsave(), la cual se utiliza para proteger estructuras de datos que son accedidas por manejadores de interrupciones en una máquina SMP (es necesario deshabilitar las interrupciones en la CPU y tomar un spinlock). También se observa que la función schedule() incrementa su tiempo de ejecución casi al doble y que _spin_lock() se incrementa en un 40 % aproximadamente, si se comparan las cantidades absolutas de muestras tomadas en estas funciones. Claramente, los costos de sincronización y scheduling se disparan más allá de lo aceptable cuando se agrega una CPU extra al procesamiento de red.

Cuadro 8.3: Profiling del kernel durante la prueba B

muestras	%	muestras	%	símbolo
113573	16.5949	1631	1.1642	_spin_lock_irqsave
77729	11.3575	61287	43.7467	__copy_to_user_ll
49851	7.2840	1816	1.2963	handle_IRQ_event
40771	5.9573	4382	3.1279	tcp_v4_rcv
25860	3.7786	5935	4.2364	tcp_rcv_established
20138	2.9425	4636	3.3092	kfree
18753	2.7401	3475	2.4805	schedule
17351	2.5353	709	0.5061	_spin_lock
15908	2.3244	619	0.4418	qdisc_restart

Para evaluar los beneficios de la afinidad de interrupciones y procesos a CPUs, se comparan los resultados entre las pruebas B y C. La primer gran diferencia se observa en el uso de CPU, incrementándose el tiempo ocioso de 15 % a 42 %, manteniendo el rendimiento de red. Los motivos de este incremento en la eficiencia pueden asociarse a una reducción en la contención por datos compartidos y un incremento en la localidad de las referencias. Por un lado, se ve en el cuadro 8.4 que el costo de sincronización se reduce (spin_lock_irqsave() desaparece de los primeros puestos), y esto puede deberse a la no interferencia entre los manejadores de interrupciones entre ellos y con las softirqs, ya que ahora el tráfico de cada NIC es procesado en una CPU distinta. Por otro lado, la afinidad de interrupciones y procesos asegura que los contextos de softirq y de la system call de recepción (donde se realiza la copia al espacio de usuario) se ejecuten en la misma CPU. Ésto, según los defensores de esta técnica, es uno de los mayores beneficios de la afinidad, ya que logra una mejora notable en el comportamiento de cache. Se puede comprobar en el cuadro 8.4 que la cantidad de muestras de fallos de cache se reduce notablemente para la función __copy_to_user_ll (de 61000 a 38000, es decir disminuye un 30 % aproximadamente).

Sin embargo también se observa un incremento notable en los cambios de

contexto ocurridos durante la prueba C (de 50000 a 200000 muestras) y el profiling muestra un incremento del 50 % en el tiempo que la función schedule() consumió de CPU. No se encontró un motivo evidente para el incremento de estos costos.

Cuadro 8.4: Profiling del kernel durante la prueba C

muestras	%	muestras	%	símbolo
71262	12.2796	38204	44.7831	__copy_to_user_ll
63721	10.9802	2039	2.3901	handle_IRQ_event
28590	4.9265	1795	2.1041	schedule
23986	4.1332	1883	2.2073	_spin_unlock
20121	3.4672	227	0.2661	_spin_lock
19769	3.4065	2567	3.0091	tcp_v4_rcv
19551	3.3690	1624	1.9037	default_idle
17150	2.9552	1631	1.9119	__switch_to
14749	2.5415	473	0.5545	qdisc_restart
11698	2.0158	1902	2.2295	try_to_wake_up

La prueba C parece ser el mejor caso en cuanto al comportamiento del kernel Linux, sin embargo, la intención del prototipo es reducir el costo de intrusión por mecanismo del sistema operativo. La prueba D, en la cual se mide la performance de la primer variante del prototipo, muestra una reducción de las interrupciones de hardware a cero y de los cambios de contexto casi al 50 % con respecto a la prueba C, aunque no se logra una reducción tan importante en el uso de CPU con respecto a la prueba B. Sin embargo, debe tenerse en cuenta que el prototipo ocupa un 50 % del poder de cómputo de la máquina (aunque no se aprovechen estos ciclos en procesamiento real) ya que utiliza una de las CPUs y nunca se devuelve el control al scheduler[2]. En una implementación real, una mejora importante sería la reconfiguración dinámica del subconjunto de CPUs dedicadas al rol de CPU-NET, dependiendo de la carga de procesamiento de red en un momento dado.

Entre otras cosas, se adjudica el incremento en el uso de CPU al hecho de que la función __copy_to_user_ll() vuelve a tener un comportamiento de cache tan perjudicial como antes, como se comprueba en el cuadro 5. También se observa un incremento del 50 % en el tiempo de ejecución de la función schedule() aunque no fue posible encontrar una explicación para este aumento. Finalmente, el profiling muestra a la función net_rx_action(), que es la función invocada constantemente por CPU-NET para realizar el procesamiento inferior; esta función

[2]Obviamente, este diseño no está pensado para ser implementado en máquinas con un número reducido de CPUs/cores.

no debe ser considerada como parte de la intrusión del sistema operativo.

Por otro lado, debe notarse que de las cinco funciones que ocupan los primeros puestos en el cuadro 8.5, tres de ellas realizan procesamiento real de protocolos. Esto parece ser una mejora con respecto a la prueba C, en la cual las cinco funciones que consumen más CPU representan intrusión del sistema operativo exclusivamente.

Cuadro 8.5: Profiling del kernel durante la prueba D

muestras	%	muestras	%	símbolo
72898	11.5857	53156	38.4372	__copy_to_user_ll
39156	6.2231	5028	3.6358	schedule
35504	5.6427	4425	3.1997	tcp_v4_rcv
27061	4.3008	4644	3.3581	try_to_wake_up
20012	3.1805	6143	4.4420	tcp_rcv_established
19172	3.0470	3123	2.2582	kfree
18464	2.9345	422	0.3051	net_rx_action
15932	2.5321	455	0.3290	qdisc_restart
13753	2.1858	736	0.5322	__switch_to
13535	2.1511	440	0.3182	local_bh_enable
13038	2.0721	3979	2.8772	eth_type_trans

Suponiendo que la CPU-NET tenía capacidad para realizar más cómputo, y no estaba siendo aprovechada (aunque se consideró como tiempo ocupado de CPU en la prueba D), se realizó la prueba E en la cual se le suma a la CPU-NET el procesamiento de la transmisión inferior. Cuando se comparan las pruebas D y E, se puede observar (en el archivo ResumenE.txt, entregado con este trabajo) que la CPU-HOST reduce su ocupación en un 10 % o, en el cuadro 1, se comprueba que se incrementa el indicador %Idle. Probablemente, esto se debe a que la CPU-HOST ahora no tiene que realizar la transmisión inferior ya que la está procesando la CPU-NET. En el profiling, ésto se manifiesta a través del incremento de tiempo de ejecución de la función desencolar(), que es la encargada de obtener los requerimientos de transmisión encolados por CPU-HOST y realizar el resto del procesamiento inferior.

Para obtener una mejor relación GHz / Gbps utilizando el prototipo, se intentaron dos pruebas con 3 placas de gigabit[3], para aumentar los requerimientos de procesamiento de la máquina. Para esto se utilizó la NIC Realtek (Eth1, 10.10.0.100), la NIC Marvell (Eth2, 192.168.0.100) y la NIC Broadcom (Eth3, 200.100.0.100).

[3]Estas pruebas fueron de cuatro minutos, con lo cual no es posible comparar cantidades absolutas de muestras con las pruebas anteriores, aunque sí se pueden comparar los porcentajes.

Cuadro 8.6: Profiling del kernel durante la prueba E

muestras	%	muestras	%	símbolo
69688	11.0974	50673	35.3457	__copy_to_user_ll
45495	7.2448	13302	9.2785	desencolar
44180	7.0354	869	0.6061	qdisc_restart
35286	5.6191	5527	3.8552	schedule
29426	4.6859	3423	2.3876	tcp_v4_rcv
27709	4.4125	4886	3.4081	try_to_wake_up
21695	3.4548	6942	4.8422	tcp_rcv_established
18298	2.9139	3857	2.6904	kfree
17770	2.8298	1221	0.8517	__tcp_select_window
14801	2.3570	482	0.3362	local_bh_enable
13366	2.1285	4103	2.8619	eth_type_trans

Sin embargo, un fenómeno que llamó la atención inmediatamente es que el rendimiento de red agregado de las tres NICs era similar al rendimiento agregado de las 2 NICs alcanzado en las pruebas anteriores, pero las CPUs no llegaban a ocuparse al 100 %, lo que indicaba que el cuello de botella se encontraba en otro componente.

Al estudiar las especificaciones de la motherboard de las máquinas en las que se realizaron las pruebas se encontró un posible motivo. El diagrama de bloques de dicha motherboard se puede observar en la figura 8.3.

La NIC Marvell está conectada directamente al chipset Nvidia (remarcado con color rojo en la figura 8.3), sin embargo la NIC Broadcom se conecta al mismo bus PCI de 32 bits a 33 MHz a los que se conectan los slots PCI de la motherboard. Como la NIC adicional que se utilizó es una placa con interfaz PCI, insertada en uno de estos slots, se compartió el ancho de banda del bus PCI que también conectaba a la NIC Broadcom (remarcado con color azul). Un bus PCI de 32 bits y a 33 MHz (como es estudió en el capítulo 2) tiene un rendimiento máximo teórico de 133 MBytes por segundo, es decir, 1064 Mbits por segundo, o 1 Gbps aproximadamente, aunque es imposible lograr un rendimiento sostenido de 1 Gbps de datos dado el overhead del funcionamiento del bus y de los protocolos de red.

En este caso, se está compartiendo 1 Gbps de capacidad teórica de transmisión entre las dos NICs. Por lo tanto, en las próximas dos pruebas se podrá observar que el rendimiento sumado de las NIC Eth1 (Realtek) y Eth3 (Broadcom) se encuentra cercano a los 700 Mbps.

Prueba X) La prueba X es una variante de la prueba C con 3 NICs. Se configuró de manera tal de lograr afinidad del procesamiento relacionado con la Eth1 y Eth3 con la CPU0, y el procesamiento de la NIC Eth2 se realiza en la

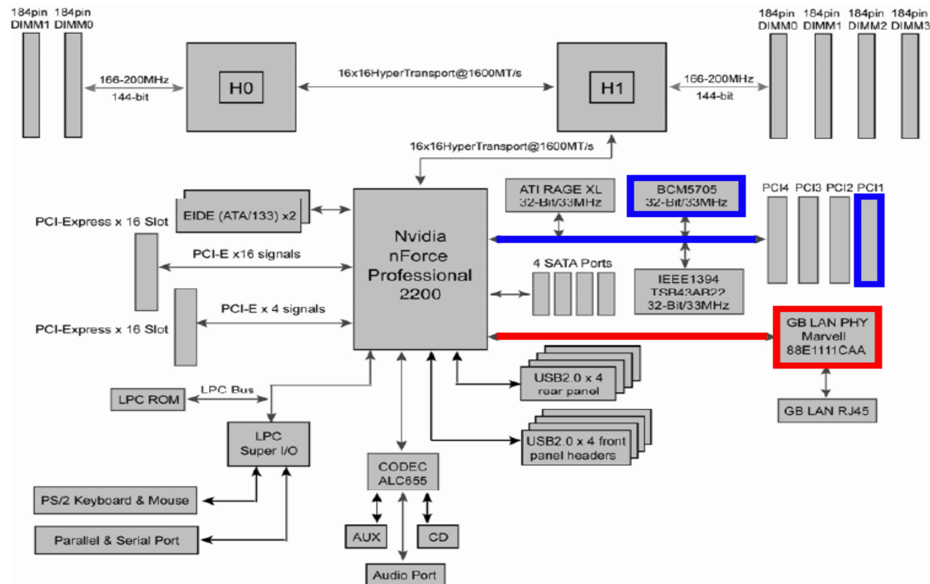

Figura 8.3: Diagrama de bloques de la motherboard utilizada en las pruebas

CPU1.

Prueba Z) Este experimento es una variante de la prueba E escalado a 3 NICs.

Cuadro 8.7: Comparación de pruebas X y Z

medidas\pruebas	X	Z
Rendimiento(mbps)	1492	1477
%Idle general	34,44	24,77
cswch/s	157186,17	99275,49
intr/s	131363	0

Al comparar las pruebas C y X, se observa que se incrementa el uso de CPU (el idle se reduce de 42 % a 34 %) aunque no se modifica el rendimiento de red. El costo adicional se puede asociar a la sincronización: la función _spin_lock() obtuvo un 4 % de las muestras en la prueba C y un 13 % de las muestras en la prueba X. Además se observa que las primeras cinco funciones siguen siendo sólo intrusión del sistema operativo.

Por otro lado, cuando se compara la prueba E con la prueba Z, el procesamiento de CPU no se modifica y esto puede indicar una mejor escalabilidad con respecto a la cantidad de NICs utilizadas. Excepto por la función

__copy_to_user_ll(), las primeras cinco funciones representan computación útil en el procesamiento de red en lugar de overhead administrativo de sincronización o manejo genérico de interrupciones.

Cuadro 8.8: Profiling del kernel durante la prueba X

muestras	%	muestras	%	símbolo
154270	13.1928	93867	49.7272	__copy_to_user_ll
153381	13.1167	1070	0.5668	_spin_lock
94636	8.0930	2845	1.5072	handle_IRQ_event
58272	4.9833	3033	1.6068	__do_IRQ
45311	3.8749	3621	1.9183	schedule
39026	3.3374	4737	2.5095	tcp_v4_rcv
30196	2.5823	1067	0.5653	qdisc_restart
26420	2.2594	3570	1.8913	__switch_to
24781	2.1192	2303	1.2200	default_idle

Si se comparan las pruebas X y Z[4], se observa que la segunda prueba reduce en más del 30 % la cantidad de cambios de contexto y lleva a cero la cantidad de interrupciones de hardware. 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 8.9: Profiling del kernel durante la prueba Z

muestras	%	muestras	%	símbolo
148053	10.3428	116811	38.1698	__copy_to_user_ll
120776	8.4372	1291	0.4219	net_rx_action
95700	6.6855	26140	8.5416	desencolar
68739	4.8020	1621	0.5297	qdisc_restart
65176	4.5531	6827	2.2308	tcp_v4_rcv
59659	4.1677	8393	2.7425	schedule
50669	3.5397	8619	2.8164	try_to_wake_up
48140	3.3630	13402	4.3793	tcp_rcv_established
43827	3.0617	9406	3.0736	kfree
36767	2.5685	2824	0.9228	__tcp_select_window
30361	2.1210	2063	0.6741	local_bh_enable
28993	2.0254	9818	3.2082	eth_type_trans

[4]En este caso se pueden comparar cantidad de muestras, ya que ambas pruebas fueron de igual duración en tiempo.

Esta prueba de concepto hace suponer que es posible reducir la intrusión del sistema operativo en el procesamiento de red. En aquellas máquinas cuya principal función es el movimiento de datos, como es el caso con un *Storage Element* en una infraestructura Grid (ver próximo capítulo), puede ser beneficial dedicar una CPU o core de manera exclusiva al procesamiento de red. Actualmente se está adaptando esta idea a una arquitectura multicore y se están realizando varias optimizaciones a la implementación que resultarán en una mejora importante en la eficiencia de procesamiento.

Capítulo 9

Conclusiones y trabajo futuro

Este trabajo describe una problemática que está captando la atención de varios grupos de investigación y que aún no presenta una solución completamente definida. Las propuestas que han demostrado tener mayor éxito son aquellas que atacan el problema a nivel de sistema completo (hardware y software) y, desde esta perspectiva, el sistema operativo es un componente esencial.

Los artículos publicados hasta ahora no dejan de resaltar que los mecanismos del sistema operativo involucrados en el procesamiento de los protocolos utilizados en nuestras comunicaciones son, básicamente, una herencia de los primeros diseños e implementaciones realizadas a fines de la década de 1970 y que existe una necesidad imperiosa de hallar una forma más eficiente de procesar el tráfico de red permitiendo potencialmente que Ethernet y TCP se acerquen a la performance, en latencia y rendimiento, ofrecida por las System Area Networks (SANs) actuales.

A lo largo de los capítulos se mencionan un conjunto de factores que parecen confluir hacia una misma dirección. Por un lado, se afirma que las redes de datos se están convirtiendo en una herramienta fundamental para el desarrollo científico y tecnológico, ya que el paradigma conocido como Network Centric Computing tiende a ser la forma predominante de cómputo. Por lo tanto, las comunicaciones influyen cada vez más en la performance de las aplicaciones.

Sin embargo, se presentan algunos obstáculos que amenazan con impedir el completo aprovechamiento de los últimos avances en redes de datos. Se observa que el requerimiento de procesamiento de las redes de alta velocidad puede superar las capacidades de una plataforma de cómputo actual y los motivos se encuentran tanto en el hardware como en el software. Ésto se debe principalmente a las diferencias relativas en el ritmo de la evolución de las tecnologías de redes, memorias, CPUs, buses de E/S y el software de base.

Con respecto a este último, se puede afirmar que, aunque se ha avanzado en el plano teórico del diseño de un kernel orientado a la E/S de alta performance, 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 altos costos, agrupados con el término intrusión del sistema operativo, surgen al utilizar sistemas operativos diseñados principalmente para tareas intensivas en computación, y no optimizados para la entrada/salida. Los mecanismos que deberían ser invocados excepcionalmente (como las interrupciones del hardware y las interrupciones de software con las que se implementan las system calls) se vuelven demasiados costosos cuando se usan con frecuencias mucho más altas que aquellas para las cuales fueron pensados.

Una forma de satisfacer los altos requerimientos del procesamiento de red es utilizar más de una CPU. Por lo tanto, se analizan las posibilidades de paralelismo en los protocolos de red y se identifica el modelo adoptado por el kernel Linux en arquitecturas SMP, el cual parece presentar problemas de escalabilidad por altos costos de sincronización, entre otras cosas. La intrusión del sistema operativo se vuelve más notable cuando se adapta un kernel monoprocesador a una arquitectura de multiprocesamiento.

Por otro lado, se supone un futuro cercano donde toda la computación estará basada en arquitecturas paralelas, y se plantea que los sistemas operativos deben ser re-diseñados teniendo en cuenta las capacidades y características específicas de esas arquitecturas. En este contexto, se afirma que en algunos casos una utilización asimétrica de los elementos de procesamiento de una máquina podría llegar a resultar más eficiente que la distribución de trabajo equitativa entre los procesadores y que se está comenzando a investigar soluciones ASMP aplicadas a varios dominios. En particular, se estudia la arquitectura de Piglet, el primer kernel que se ejecuta de manera activa en CPUs dedicadas, buscando reducir la intrusión del sistema operativo.

Siguiendo estas ideas, se implementó un prototipo para este trabajo que modifica la forma de procesar paquetes de red en el kernel Linux 2.6 en una máquina SMP, se realizaron ciertas pruebas comparativas de performance y se identificaron las fuentes de overhead gracias al uso de profiling estadístico. Linux es el kernel monolítico más utilizado en los sistemas operativos libres y por lo tanto representa una gran oportunidad para estudiar y proponer optimizaciones. Además es altamente modular y existe gran cantidad de documentación relacionada, lo que facilita realizar modificaciones a su código. [1]

Se demostró que es posible reducir la intrusión del sistema operativo gene-

[1] Actualmente, el kernel linux 2.6 ha alcanzado un nivel de estabilidad, robustez y flexibilidad notables. provee soporte para más de veinte arquitecturas, incluyendo todo tipo de máquina NUMA, SMP y dispositivos embebidos. Mejoró considerablemente gran parte de los subsistemas con respecto al kernel 2.4 teniendo como principal objetivo la escalabilidad.

rada por el procesamiento de E/S de red, si se construye un kernel pensado para esta tarea. Evitar el uso de interrupciones de software y de hardware, dedicando un elemento de procesamiento para sondear el estado del hardware de red y atender los posibles requerimientos que los procesos de usuario envían a través de la memoria compartida parece ser una alternativa viable, siempre que la carga de red lo justifique.

9.1 Trabajo a futuro

Como se mencionó en el capítulo 8, el prototipo no delega las copias entre el espacio del kernel y el espacio de usuario a la CPU dedicada, y ésta es una de las operaciones más costosas en el procesamiento de red. En el capítulo 7 se mencionaron algunas iniciativas de mejorar la performance de la interfaz entre el kernel y los procesos de usuario, a través del uso de memoria compartida, system calls asincrónicas y señalización por eventos. Recientemente, han aparecido prototipos básicos que implementan algunas de estas técnicas en el kernel Linux y es posible combinarlas con el prototipo propuesto en esta tesis.

Otra mejora importante al prototipo es la de implementar la posibilidad de reconfiguración dinámica del subconjunto de CPUs dedicadas al rol de CPU-NET, dependiendo de la carga de procesamiento de red en un momento dado. Como se mencionó, la interrupción del timer local APIC se mantiene habilitada en la CPU-NET, lo que deja abierta la posibilidad de disparar eventos periódicos, como pueden ser evaluar la situación y realizar reconfiguración automática. Ésto se puede implementar asignando un thread del kernel a cada CPU en el momento del inicio de la máquina, y despertando o durmiendo threads a medida que son necesarios.

Finalmente, se planea estudiar las posibles aplicaciones de este trabajo en un entorno de Grid Computing. Es posible que algunos componentes de un middleware de Grid puedan beneficiarse de un sistema operativo que reduce los costos de mover grandes cantidades de datos en la red. Por ejemplo, en un entorno en producción como el que representa el acelerador de partículas más grande del mundo en el CERN (http://lhc.web.cern.ch/lhc/), se distribuirán diariamente decenas de terabytes de información en la Grid formada por diferentes laboratorios en el mundo. Estas transferencias utilizan el protocolo gridFTP y se realizan entre los componentes conocidos como *Storage Element*, de los diferentes sitios de la Grid. Tal vez, ejecutar un kernel optimizado para la E/S de red como base de un Storage Element incremente la eficiencia y/o rendimiento de las transferencias.

Bibliografía

[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] A. Grover and C. Leech. Accelerating Network Receive Processing. Proceedings of the Linux Symposium.

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

[14] Intel. Accelerating High-Speed Networking with Intel I/O Acceleration Technology.

[15] Intel. Server Network I/O Acceleration. Fundamental to de Data Center of the Future.

[16] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. Proc. ACM Communications Architectures and Protocols Conf. (SIGCOMM), San Francisco, CA, pp. 259-269, Sept. 1993.

[17] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. ACM Transactions on Computer Systems, 1995.

[18] Philip J. Mucci. Memory Bandwidth and the Performance of Scientific Applications. Innovative Computing Laboratory, University of Tennessee. June, 2004.

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

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

[21] Vivek S. Pai, Peter Druschel, and Willy Zwaenep. Flash: An e cient and portable Web server. Department of Electrical and Computer Engineering and Department of Computer Science. Rice University.

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

[23] 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.

[24] 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.

[25] 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.

[26] P. Wan and Z. Liu. Operating System Support for High-Performance Networking, A Survey. Computer and Information Science Department. Indiana University Purdue University Indianapolis.

[27] Wikipedia. Flynn Taxonomy. http://en.wikipedia.org/wiki/Flynn_taxonomy.

[28] Wikipedia. Remote Direct Memory Access. http://en.wikipedia.org/wiki/Rdma.

[29] Wikipedia. TCP Offload Engine. http://en.wikipedia.org/wiki/TCP/IP_offload_engine.