A Light-Weight Virtual Machine Monitor for Blue Gene/P

CARLOS MARTÍN FLORES GONZÁLEZ, Carné: 2015183528

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

Para la supercomputadora Blue Gene/P (BG/P) en producción IBM provee un kernel liviano llamado *Compute Node Kernel*(CNK). CNK corre tareas masivamente en paralelo usando un thread por núcleo (*single-thread-per-core*). Como otros kernels livianos, CNK soporta un subconjunto de interfaces de aplicación estandarizadas(POSIX), facilitando el desarrollo de aplicaciones(POSIX-like) dedicadas para supercomputadoras. Sin embargo CNK no es totalmente compatible con POSIX. CNK también soporta entrada/salida solamente a través de *function-shipping* a nodos de entrada/salida. El modelo simplificado de CNK es una buena opción para el conjunto actual de aplicaciones de alto rendimiento en BG/P, sin embargo las aplicaciones de alto rendimiento hoy en día están empezando a escalar hace sistemas Exascale[1] de dimensiones realmente globales y que abarcan varias instituciones, compañías e inclusive países. El soporte restringido de interfaces estandarizadas de aplicación de los kernels livianos en general y CNK en particular hace que esta situación se convierta en un cuello de botella en el desarrollo de aplicaciones de computación de alto rendimiento. En este artículo se explora una alternativa, un sistema operativo (SO) híbrido para la BG/P: un monitor de máquina virtual(VMM) basado en μ−kernel. En la capa más baja, el kernel, se corre un μ−kernel que provee un pequeño conjunto de primitivas de SO para construcción de aplicaciones de alto rendimiento personalizadas y servicios de nivel de usuario. Luego se construye un VMM de nivel de usuario que virtualiza completamente la plataforme BG/P y permite que SO arbitrarios se ejecuten en compartimentos virtualizados. Beneficios: (1) provee compatibilidad con el hardware de la BG/P, lo que le permite a los programadores usar el SO que ellos necesiten en su aplicación en particular como una librería. (2) El μ−kernel que se propone se asemeja mucho a los enfoques de kernels simplificados en el sentido que reducen la cantidad de funcionalidad del kernel a gestión básica de recursos y comunicación. Sin embargo, las arquitecturas de μ−kernel y VMM también impactan en la eficiencia ya que incrementan la interacción kernel-usuario y agrega otra capa de indirección dentro del sistema.

System Overview. El bloque básico de un BG/P es un *computer node*, el cual está compuesto de una PowerPC quadcore embebido, cinco interfaces de red, un controlador DDR2 y 2 o 4GB de RAM integrado todo en un chip. La implementación BG/P usa una versión del μ−kernel L4 -L4Ka::Pistachio. L4 se diferencia de VMMs tradicionales porque ofrece un conjunto limitado de abstracciones de SO para forzar seguridad y una ejecución segura. La funcionalidad del VMM real está implementada como una aplicación de nivel de usuario por fuera del kernel privilegiado. El mecanismo básico del VMM basado en L4 es el siguiente: L4 actua como un sistema de mensajería segura propagando instrucciones sensitivas del huésped (*guest*) al VMM de nivel de usuario. VMM a su vez, decodifica la instrucción y la emula apropiadamente y luego responde con un *fault reply message* que le dice a L4 que actualice el contexto de la máquina virtual(VM) *guest* y luego retome la ejecución. Configuraciones híbridas son posibles: una aplicación puede iniciar dentro del kernel de una VM *guest* y luego podría escoger aplicar alguna librería nativa para computación de alto rendimiento.

A Micro-Kernel With Virtualization Capabilities. Modelo de virtualización: (1) L4 threads para virtualizar el procesador (2) Mecanismos de mapeo de memoria de L4 para proveer y gestionar la memoria física del *guest*. (3) *Interprocess Communication* (IPC) para permitir la emulación de instrucciones sensitivas a través del VMM de nivel de usuario. *Virtual PowerPC Processor:* L4 virtualiza núcleos(*cores*) mapeando cada CPU virtual (vCPU) en un thread dedicado. Los threads

[1]La Computación Exascale hace referencia a los sistemas de computación capaces de realizar un mínimo de un exaFlops, es decir, 10^{18} cálculos por segundo.

son entidades programables(*schedulable*) y los vCPUS son tratados igual. Los núcleos de BG/P están basados en la arquitectura PowerPC 440 que no tiene soporte dedicado para facilitar o acelerar la virtualización, pero PowerPC es más amigable para virtualización que x86 porque suporta virtualización basada en *traps* e instrucciones de largos fijos que simplifican la decodificación y emaulación de instucciones sensitivas. L4 emplea este estilo de virtualización de *trap-and-emulate* comprimiendo niveles de privilegio en PowerPC. L4 como tal corre en modo supervisor, mientras que usuario y *guest* corren en modo usuario. Aplicaciones *guest* corren sin disturbios pero cuando el kernel *guest* genera una instrucción sensitiva, el procesador causa un *trap* en L4. *IPC-based Virtualization:* L4 como tal no emula todas las instrucciones sensitivas por si mismo. A menos que la instrucción esté relacionada con TLB virtual o que pueda ser manejada rápidamente, le deja la emulación al VMM de nivel de usuario. Protocolo de virtualización implementado por el IPC de L4: *guest-trap* → emulación de VMM → ciclo de reinicio del *guest*. L4 maneja *guest traps* en la misma forma que maneja fallos de página y excepciones, sintetizando un mensaje IPC de fallo en nombre del *guest* hacia un manejador de excepciones designado por vCPU. *Virtualized Memory Management:* Un VMM tiene que soportar dos niveles de traducción, de *guest-virtual* a *guest-physical* y de *guest-physical* a *host physical memory*. Como PowerPC 440 solamente soporta un nivel de traducción, el VMM tiene que unir los dos nivels cuando inserta traducciones en el hardware del TLB, efectivamente traduciendo direcciones *guest-virtual* a *host-physical*. *Virtual Physical Memory:* L4 trata el espacio de direcciones físicas del *guest* VM como un espacio normal de direcciones y exporta el establecimiento de traducciones dentro de ese espacio de direcciones a un paginador de nivel de usuario (que en este caso es el VMM de nivel de usuario). Cuando un *guest* sufre un intento fallido de TLB, el manejador de intentos fallidos de L4 inspecciona sus estructuras de datos de kernel para encontrar si el intento fallido ocurrió por una traducción fallida *guest-physical* a *host-physical*. Este es el caso si el VMM no tiene aun mapeado la memoria física dentro del especial de direcciones físicas del *guest* VM. Si es asi, L4 sintetiza un *page fault IPC message* al paginador del VMM de nivel de usuario en nombre del *guest* que está fallando, solicitando que se le de servicio a la falla. Cuando el VMM encuentra que el fallo de página física del *guest* es válida, responde con un mensaje de mapeo que causará que L4 inserte un mapeo válido en el TLB y que se retome la ejecución. *Virtual TLB:* BG/P no tiene soporte para emular traducciones de direcciones virtuales y se usa una solución de software llamada TLB virtual. L4 provee la noción de un TLB virtual al cual el *guest* tiene acceso a través de instrucciones normales (*trapped*) de hardware para gestión de TLB. La solución para la memoria virtual del *guest* es un por medio de instrucciones internas de L4. Cuando el *guest* kernel accede al TLB virtual, instrucciones de emulación internas de L4 guardan las entradas dentro de una estructura de datos de TLB virtual por VM. En un intento fallido de TLB, el manejador de intentos fallidos de L4 parsea esa estructura de datos para encontrar si el *guest* ha insertado un mapeo válido de TLB dentro del su TLB virtual para la dirección fallida dada. Si no es asi, inyecta un intento fallido de TLB dentro del *guest* VM para hacer que el intento fallido sea manejado por el kernel *guest*. Si el TLB virtual de hecho contiene una entrada válida, L4 verifica su base de datos de mapeos para encontrar si el intento fallido ocurrió de *guest-physical* a *host-physical*. Si esa traducción es válida también, L4 inserta el mapeo resultante *guest-virtual* a *host-physical* en el TLB del hardware y retoma la ejecución. *Virtual Address Protection:* un VMM debe virtualizar no solo el motor de traducción del TLB sino también su características de protección. La virtualización requiere lógicamente dos niveles, permitiendo al *guest* usar los bits e identificadores de protección del TLB virtual en la misma forma que en hardware nativo pero, en el segundo nivel, también permite a L4 y a su espacio de direcciones de nivel de usuario proteger sus datos de que sean accedidos por el kernel *guest* y aplicaciones. Para facilitar la virtualización *trap-and-emulate*, ambos, el kernel *guest* y las aplicaciones corren en modo usuario. L4 pone el kernel *guest* y de usuario en un segundo espacio de traducciones, y mantiente el primer espacio de traducción reservado para él mismo, el VMM y las aplicaciones nativas L4. El procesador cambia automáticamente al primer espacio de traducción cuando una interrupción o una *trap* ocurre, entrando directamente a L4 con manejadores de interrupciones y *traps*. Para leer memoria del *guest* cuando se decodifica una instrucción sensitiva, L4 cambia temporalmente el espacio de traducciones para recuperación de datos, mientras retiene el espacio para instrucciones. En total la solución permite al *guest* recibir un espacio de direcciones vacío, pero en cualquier excepción, el procesador cambia a un espacio de direcciones propiedad del VMM. El máximo objetivo es el de reducir el número de TLB *flushes* para reforzar la protección en cambios usuario-a-kernel. Se esfuerza en hacer esta transición rápida para lograr un buen rendimiento de *system call*; por lo tanto se deshabilita el espacio de mapeios complementamente y se hace *flush* del TLB del hardware en cada cambio de espacio de direcciones del *guest*. Se usa la dirección 0 para mapeos que son accesibles de *guest* usuario y *guest* kernel. Se usa espacio de direcciones 1 para mapeos que son solo accesibles por *guest*-usuario y el espacio de direcciones 2 para mapeos que son accesibles por el modo *guest*-kernel. Un cambio(*switch*) de nivel privilegiado de *guest*-usuario a *guest*-kernel por lo tanto solo requiere de actualización del identificador del espacio de direcciones de 1(mapeos de modo usuario) a 2(mapeos de modo kernel). *Interrupt Virtualization:* GB/P provee un controlador personalizado

de interrupciones llamado *Blue Gene Interrupt Controller*(BIC), el cual reúne y entrega señales de dispositivos a los núcleos como interrupciones o *machine check exceptions*. El BIC soporta un gran número de interrupciones que son organizadas en grupos y tienen diferentes tipo de prioridad y formas de ser entregadas. Soporta entrega de interrupciones a otros núcleos tambien como interrupciones núcleo-a-núcleo. El VMM de nivel de usuario usa el soporte en interrupciones que provee L4 para recibir y reconocer interrupciones para los dispositivos BG/P. Para inyectar interrupciones virtuales en el *guest*, el VMM modifica el estado de vCPU ya sea por medio de un llamado al sistema o bien llevando la actualización del estado a la *virtualization fault reply*, en el caso de que el VM *guest* está esperando por el VMM cuando una interrupción ocurre.

User-Level VMM:. La capa de interfaz que traduce invocaciones del API de virtualización en invocaciones del API de la arquitectura subyacente de L4. El VMM principalmente consiste de un servidor ejecutando un ciclo de IPC, esperando por algun mensaje IPC entrante de un VM *guest* que falla. *Emulating Sensitive Instructions:* Para emular instrucciones sensitivas con la llegada de un *virtualization fault IPC*, el VMM decodifica la instrucción y sus parámetros basado en un puntero al *program counter* y un archivo de registros de propósito general del VM *guest*, los cuales están almacenados dentro del mensaje IPC que fue enviado desde L4 en nombre de la VM que lanzó la trampa. Por conveniencia, L4 también pasa el valor del program counter, que es, la instrucción que lanzó el *trap*. *Virtual Physical Memory:* para el VMM, paginar un *guest* con memoria física virtualizada es similar a la paginación normal de nivel de usuario en sistemas L4: cuando el *guest* sufre un intento fallido de TLB, L4 envía un *page fault IPC* conteniendo la instrucción que falla, la dirección y otros estados de TLB (virtuales) necesarios para procesar la falla. En su implementación, el VMM organiza la memoria física del *guest* en segmentos lineares. Asi, cuando se maneja una falla, el VMM verifica si la dirección si la dirección física del *guest* accesada está dentro de los límites del segmento. Si es así responde con un mensaje IPC de mapeo qeu causa L4 a insertar el mapeo correspondiente en su base de datos y en el TLB del hardware. *Device Virtualization:* El VMM provee modelos virtuales para el BIC y para los dispositivos de *collective* y *torus network*. La emulación del BIC es un tanto simple: el VMM intercepta todo los accesos al dispositivo BIC mapeado en memoria y lo emula usando los mecanismos de L4 para manejo de interrupciones externas e inyección de eventos. *Collective Network:* es un árbol binario sobreconectado que abarca la instalación por completo. El colectivo es un medio uno-a-todos para operaciones de transmisión o reducción, con soporte para filtrado por nodo específico y un complejo esquema de ruteo que permite particionar la red para incrementar el total de ancho de banda. El VMM emula los registros de control del dispositivo(DCR) usado para configurar el dispositivo de *collective network*. Las instrucciones correspondientes son sensitivas y directamente atrapadas por L4 y el VMM. El VMM se registra así mismo en L4 como un manejador de interrupciones para todas las interrupciones de dispositivos *collective network*, lo que causará que L4 emita un mensaje IPC de interrupción al VMM cuando el dispositivo físico dispare uno de sus interrupciones de hardware. *Torus:* cada *compute node* es parte de una red toroidal en 3D que abarca toda la instalación. En cada nodo, el *torus device* tiene enlaces de entrada y salida para cada uno de sus 6 vecinos. El *torus* provee dos interfaces de transmisión, una normal basada en buffer y otra basada en accesos de memoria directo remoto (rDMA). El VMM provee una versión virtualizada del dispositivo de *torus* y atrapa y emula todo los accesos a los registros del dispositivo de *torus*. Las instrucciones DCR son atrapadas por directamente en L4. El VMM se registra a si mismo para interrupciones de *torus* fisicas y las entrega a los *guest* como sea necesario.

Initial Evaluation: Hubo un mayor enfoque en la funcionalidad que en el rendimiento. *Guest OS support:* El VMM basado en L4 soporta la ejecución de SO huésped en GB/P como CNK o ZeptoOS. Se ha verificado la instalación con Kittyhawk Linux. El VMM permite uno o más instancias de Kittyhawk Linux corran como VM. Kittyhawk corre sin modificación alguna lo que quiere decir que el mismo binario que es soportado por BG/P también corre por encima del VMM. Solamente se soporta *guests* uni-procesador, sin embargo, como L4 soporta multiprocesador, los vCPUs de los *guests* pueden ser programados en cada uno de los cuatro núcleos físicos de un nodo BG. *Initial Benchamark Results:* En el primer experimento se compiló un proyecto de código pequeño en la versión virtualizada de Kittyhawk Linux. Luego se usó una herramienta de debug de L4 con rastreo interno de eventos para encontrar rutas de código de VM frecuentemente ejecutadas. En esta configuración la compilación duró 126s contra los 3s que se duró en la versión nativa de Kittyhawk Linux. Se notó que el número de IPCs -esto es, el número de salidas del VM que involucran el VMM de nivel de usuario- es relativamente bajo, lo que significa que L4 maneja la mayoría de las salidas del *guest* internamente. También en el experimento mostró que un algo número de intentos fallidos de TLB y de instrucciones relacionadas con TLB, lo que indica que el subsistema de memoria virtualizada es un cuello de botella en la implementación. En el segundo experimento, se midió el *throughput* y la latencia entre dos *compute nodes*. Paquetes fueron enviados al *torus interconnect* por medio del modulo controlador de Ethernet de Kittyhawk Linux. Para comparación se corrió los experimentos para ambos el nativo y virtualizado, cada uno corriendo en un *compute node*.

También se uso la herramienta `netperf` para probar el *throughput* y la latencia. En los resultados se mostró que la capa de virtualización plantea un *overhead* significativo al rendimiento de la red Ethernet.

1. ¿CUÁL ES EL PROBLEMA QUE PLANTEA EL *PAPER*?
2. ¿POR QUÉ EL PROBLEMA ES INTERESANTE O IMPORTANTE?
3. ¿QUÉ OTRAS SOLUCIONES SE HAN INTENTADO PARA RESOLVER ESTE PROBLEMA?
4. ¿CUÁL ES LA SOLUCIÓN PROPUESTA POR LOS AUTORES?
5. ¿QUÉ TAN EXITOSA ES ESTA SOLUCIÓN?