WICC 2009

Análisis de compresión de un sistema de archivos sobre memoria Flash usando OLPC[1]

Lic. Matías Zabaljáuregui
Mg. Lia Molinari
Lic. Javier Díaz (Director del LINTI)
{matiasz, lmolinari, javierd}@info.unlp.edu.ar
Calle 50 y 120 – 2do Piso TE: 0221-4223528. FAX: 0221-427-3235
L.I.N.T.I.. - Facultad de Informática – U.N.L.P.

Resumen

Las ventajas de las técnicas de compresión al vuelo (compresión on the fly) se han estudiado desde hace ya una década para su posible utilización en los diferentes subsistemas del kernel de un sistema operativo. Se han planteado soluciones de compresión transparente para el usuario de datos almacenados en memoria o disco y datos enviados por la red, con resultados notables.
Una alternativa para implementar esta idea de manera transparente para el usuario y sin tener que realizar modificaciones importantes al diseño del sistema operativo es incluir funcionalidad extra a un sistema de archivos, de manera tal que cuando el usuario escriba datos al sistema de archivos, éste los comprima justo antes de volcarlos en el disco y cuando el usuario intente recuperar sus datos, el sistema de archivos los descomprima.
Este trabajo estudia las ventajas de aplicar esta última idea a un sistema de archivos especialmente diseñado para memorias Flash, para el proyecto OLPC.

1 Sobre la compresión transparente en sistemas de archivos

Las técnicas de compresión de datos han ayudado a hacer un uso más eficiente del espacio en disco, el ancho de banda de red y otros recursos similares.
La mayoría de las aplicaciones de compresión requieren una acción explícita por parte del usuario para comprimir y descomprimir archivos de datos. Sin embargo, hay algunos sistemas en los cuales la compresión y descompresión de archivos es hecha de manera transparente por el sistema operativo. Un archivo comprimido requiere menos sectores en el disco y además, el tiempo adicional requerido para la compresión y descompresión de archivos de datos puede compensarse con el tiempo ganado al reducir el acceso al disco (uno de los dispositivos más lentos en una computadora).
Existen un conjunto de proyectos que han intentado incluir alguna forma de compresión de datos como funcionalidad transparente del sistema de archivos de un sistema operativo. Estos proyectos, en general, coinciden con las características buscadas en sus diseños: mantenimiento de performance, transparencia al usuario (por ejemplo, el tamaño y otros atributos del archivo visibles para el usuario, deberían permanecer igual que los del archivo sin comprimir), compatibilidad binaria con los programas existentes. Otro importante punto es a qué nivel se realiza la compresión: a nivel de archivo o a nivel de bloque lógico.

[1] OLPC: One Laptop Per Child

La compresión a nivel de archivo lleva a una mayor degradación de la performance cuando se leen y se escriben pequeñas partes del archivo, ya que el archivo entero necesita ser comprimido para cada escritura y necesita ser descomprimido para cada lectura. La ventaja de performance de comprimir datos al nivel de bloque lógico supera ampliamente la dificultad asociada con su implementación, por lo tanto, se suele optar por esta alternativa. Otra de las ventajas de usar bloques lógicos como la unidad de compresión es que si un bloque de disco se corrompe, sólo los datos correspondientes a ese bloque lógico son perdidos. Si el archivo fuera la unidad de compresión, todos los datos del archivo se perderían si se corrompe sólo un bloque. Una desventaja importante de este diseño es la necesidad de modificar una parte del Virtual File System [1] del kernel ya que aunque el tamaño de bloque lógico siempre es un múltiplo entero del bloque físico, el número de bloques físicos requeridos para almacenar un bloque lógico depende de cuanto sea comprimido ese bloque de datos.
Un objetivo deseable de un sistema de archivos comprimido es que soporte múltiples técnicas de compresión dinámicamente. La elección de la técnica particular de compresión podría hacerla el administrador del sistema y el usuario. El administrador del sistema puede elegir la técnica de compresión a ser usada en el momento de montar el sistema de archivos. Por otro lado, los beneficios de la compresión no son significativos en el caso de archivos muy pequeños, por lo que se podría comprimir sólo archivos que tengan un tamaño mayor que un determinado umbral. Valga como ejemplo el caso de los directorios, que, como suelen ser archivos de tamaño pequeño, y más frecuentemente accedidos, podría ser conveniente no comprimirlos.

1.1 Sistemas de archivos estructurados como Log

La idea fundamental de un sistema de archivos estructurado como log es mejorar la performance de escritura almacenando temporalmente todas las modificaciones en el file cache y luego escribiendo todos los cambios secuencialmente en una única operación de transferencia al disco. La información escrita al disco incluye bloques de datos de archivos, atributos de archivos, bloques índices, directorios y casi toda la información necesaria para administrar un sistema de archivos.
Para entender como funciona un sistema de archivos estructurado como log, es necesario explicar dos ideas básicas. La primera relacionada con almacenar el sistema de archivos completo como un árbol, proveyendo de esta forma un único punto de inicio para todo direccionamiento. La segunda involucra el dividir el dispositivo de almacenamiento en segmentos relativamente grandes en los cuales se escriben los nuevos datos.
Generalmente los sistemas de archivos Unix estructuran sus datos como un árbol, con el directorio root como raíz. Sin embargo, ciertos metadatos del sistema de archivos, como la tabla de inodos y el mapa de bloques de disco libres, no son direccionados como parte del árbol. En su lugar, estos elementos tienen ubicaciones fijas en el dispositivo de almacenamiento.
Es posible almacenar los metadatos en archivos especiales para lograr un sistema de archivos completamente direccionado como un árbol y de esta forma el único dato que tiene una ubicación fija en el dispositivo es el puntero a la raíz del árbol. La raíz entonces puede apuntar, posiblemente vía bloques indirectos, al archivo especial que contiene todos los inodos. Uno de estos inodos puede ser el que identifica a un archivo que contiene el mapa de bloques libres, otro inodo puede contener el directorio root, y siguiendo este patrón puede encontrarse todos los datos y metadatos del sistema de archivos.
Teniendo el sistema de archivos estructurado como un árbol, se dispone de gran flexibilidad para almacenar los datos y metadatos en cualquier parte del dispositivo. Como no hay datos con ubicaciones fijas, cualquier cosa puede almacenarse en cualquier lugar del disco. Esto nos lleva al segundo aspecto fundamental de un sistema de archivos estructurado como log: los segmentos.
El dispositivo se divide lógicamente en un número de segmentos relativamente grandes. Cuando se necesita escribir datos al disco, se busca un segmento no usado y se escriben todos los datos junto

con los metadatos apropiados. Todos estos datos son escritos secuencialmente en uno o más segmentos y finalmente la dirección del inodo raíz se almacena en una ubicación fija del dispositivo. Desde ese inodo puede recorrerse completamente el estado actualizado del sistema de archivos.

1.2 Compresión en un sistema de archivos estructurado como log

La compresión transparente de datos no se ha generalizado en los sistemas de archivos convencionales, principalmente por dos razones.

Por un lado, los algoritmos de compresión no proveen una compresión uniforme de todos los datos. Cuando un bloque de un archivo es sobrescrito, los nuevos datos pueden ser comprimidos con otra relación con respecto a los datos anteriores, por lo tanto, el sistema no podría simplemente sobrescribir los bloques originales. Si los nuevos datos comprimidos son más grandes que los anteriores, deben ser escritos en otra ubicación; si son más pequeños, el sistema de archivos puede o bien reusar el espacio original de los datos (desaprovechando parte del mismo) o liberar el espacio y alocar un espacio nuevo más pequeño. En cualquier caso, el espacio de disco tiende a volverse fragmentado, lo cual reduce la efectividad de la compresión.

Por otro lado, los mejores algoritmos de compresión son adaptativos, es decir que usan patrones descubiertos en una parte de un bloque para hacer un trabajo más efectivo en comprimir la información de otras partes.

Estos algoritmos trabajan mejor en bloques de datos de gran tamaño, en lugar de bloques pequeños. La tendencia en la variación en la relación de compresión, aunque para diferentes algoritmos y diferentes datos, suele ser la misma: los bloques más grandes logran mejor relación de compresión [10].

Sin embargo, es difícil lograr que bloques de datos lo suficientemente grandes sean comprimidos a la vez. La mayoría de los sistemas de archivos usan tamaños de bloques que son demasiado pequeños para una compresión de datos efectiva, e incrementar el tamaño del bloque significaría una gran desperdicio de espacio por fragmentación. Además, comprimir múltiples bloques simultáneamente y de manera eficiente resulta difícil ya que los bloques de disco adyacentes no siempre se escriben al mismo tiempo. Comprimir archivos completos tampoco presenta una solución en la práctica ya que en gran parte de los sistemas de archivos la mayoría de los archivos son de sólo algunos kilobytes, además de los problemas de eficiencia mencionados anteriormente.

Afortunadamente, en un sistema de archivos estructurado como log, la estructura de datos principal en el disco es un log escrito secuencialmente.

Todos los datos nuevos, incluyendo las modificaciones a los archivos existentes, son escritos al final del log. Esta técnica fue introducida por Rosenblum y Ousterhout en un sistema de archivos llamado Sprite LFS[9]. El objetivo principal de LFS es proveer performance eliminando las búsquedas de disco en las escrituras.

Este tipo de sistema de archivos es ideal para agregar compresión transparente de los datos. Simplemente se comprime el log a medida que se va escribiendo. Como no se sobrescriben bloques, no es necesario realizar acciones especiales cuando los nuevos datos se comprimen con otra relación y, como los bloques son escritos secuencialmente en el disco, se elimina la fragmentación.

2 DEFLATE/zlib

Entre otros [11], DEFLATE es un algoritmo de compresión de datos sin pérdida que usa una combinación del método LZ77 y la codificación de Huffman. Fue definido originalmente por Phil Katz para la versión 2 de su herramienta PKZIP y luego fue especificado formalmente en la RFC 1951.

zlib[13] es una librería de compresión open-source que surgió como una abstracción del algoritmo DEFLATE utilizado en el programa de compresión de archivos gzip. La primer versión fue publicada en 1995 para ser utilizada en la librería de imágenes libpng.

Existen cientos de aplicaciones para sistemas operativos tipo UNIX que se basan en este algoritmo para comprimir sus datos y su uso en otras plataformas, como Palm OS y otros sistemas embebidos, está creciendo notablemente, ya que que su código es portable y tiene un footprint de memoria principal relativamente pequeño.
La compresión se logra a través de dos pasos:
• La búsqueda de strings duplicados y su reemplazo por punteros (pares distancia, tamaño), utilizando el algoritmo basado en diccionario LZ77.
• El reemplazo de los símbolos de salida del proceso anterior con nuevas codificaciones basadas en la frecuencia de uso, utilizando la codificación Huffman.
En DEFLATE se presenta una sutil modificación del método Huffman para evitar tener que almacenar y/o transmitir el árbol generado por el algoritmo. En la variación utilizada por el estándar DEFLATE, hay dos reglas adicionales:
• Los elementos que tengan códigos más cortos se colocan a la izquierda de aquellos con códigos más extensos
• Entre los elementos con códigos de igual longitud, aquellos que están primeros en el orden del conjunto de elementos se ubican a la izquierda.
Cuando se imponen estas dos restricciones en la construcción del árbol de Huffman, existe un único árbol para cada conjunto de elementos y sus respectivas longitudes de código. Estas longitudes son los únicos datos requeridos para reconstruir el árbol y, por lo tanto, lo único que se almacenará y/o transmitirá.
Por otro lado, el compresor de DEFLATE ofrece gran flexibilidad a la hora de comprimir los datos, pues hay tres modos de compresión disponibles.

3 La tecnología Flash y JFFS2

Flash es una tecnología de almacenamiento cuya utilización está creciendo notablemente en sistemas embebidos porque provee almacenamiento basado en estado sólido con alta densidad y confiabilidad, a un costo relativamente bajo.
Las memorias Flash son de tipo no volátil, una característica muy valorada para la gran cantidad de aplicaciones en las que se emplea este tipo de memoria, principalmente pequeños dispositivos basados en el uso de baterías como teléfonos móviles, PDA, pequeños electrodomésticos, cámaras de fotos digitales, reproductores portátiles de audio, etc. [2] [3]
Los objetivos del diseño de Journaling Flash File System (JFFS) [12] [8] están determinados en gran medida por las características de la tecnología Flash y de los dispositivos en los que se supone que será utilizado. Como los dispositivos embebidos y alimentados por baterías usualmente son tratados por los usuarios como simples electrodomésticos o aparatos electrónicos, es necesario asegurar una operación confiable frente a interrupciones en la alimentación eléctrica.
El JFFS original es un sistema de archivos basado en logs puro.

4 Las pruebas realizadas con OLPC

Las pruebas de performance fueron realizadas sobre una plataforma embebida que utiliza un chip Flash NAND de 512 MB como dispositivo de almacenamiento secundario. El hardware de prueba consiste en la placa base de la primer generación de prototipos del proyecto One Laptop Per Child gestado en Media Labs del MIT[7].
Las pruebas consistieron en medir el tiempo que consumen las operaciones de lectura y escritura de bloques de datos en la memoria Flash de la OLPC utilizando JFFS2 con y sin compresión, y en calcular las latencias introducidas por las funciones de compresión y descompresión implementadas en el kernel. De esta forma, se logró demostrar que aunque la compresión introduce una latencia no

despreciable, el tiempo final de la operación de escritura resulta menor que en el caso de no utilizar compresión.
Luego de analizar distintas herramientas de benchmarks para sistemas de archivos, se realizó un primer juego de pruebas con aquella que ofrecía la mayor flexibilidad para medir latencias en las operaciones de lectura y escritura de bloques de datos.
Iozone[6] es un instrumento open-source que permite medir el rendimiento y latencia en varias de las operaciones posibles sobre un sistema de archivos. Sin embargo, al realizar las pruebas se alcanzaban relaciones de compresión que excedían los promedios normales de los algoritmos de compresi´on utilizados. Al estudiar el código fuente de Iozone, se pudo comprobar que éste genera los datos a ser escritos en el sistema de archivos utilizando un único patrón de caracteres que se repite hasta completar un archivo del tamaño deseado por el usuario.
Para las pruebas realizadas para este trabajo, se escribieron dos programas específicos que miden el tiempo consumido por cada operación de bloque escrito o leído. [4] [5]
Por otro lado, para estudiar la latencia introducida por las funciones de compresión y descompresión, en primer instancia se buscaron mecanismos ofrecidos por el kernel linux para medir tiempos. Fue necesario realizar la medición en espacio de kernel para poder aislar únicamente la latencia de las funciones que implementan la compresión y descompresión de los datos, sin incluir el tiempo consumido en otras tareas realizadas por las system calls read() y write().

5 Los resultados obtenidos

Aunque se disponen de los datos detallados para la transferencia de cada bloque, a continuación se realiza el análisis estudiando las latencias promedios para minimizar el efecto del read-ahead en el caso de las lecturas y la activación del garbage collector de jffs2 en el caso de las escrituras.
Como se puede observar en el Cuadro 1, en el caso de la configuración de jffs2 sin comprimir, el tiempo promedio que insume escribir un bloque de 4096 bytes es de 8240 microsegundos, mientras que la latencia producida al invocar a la función jffs2 compress(), es de 1,5 microsegundos. Como en este caso no se están comprimiendo los datos, la invocación a la función virtual de compresión simplemente retorna sin realizar ninguna acción. Por lo tanto, los 8240 microsegundos incluyen el tiempo de ejecución de la system call más el tiempo de acceso y transferencia de datos al dispositivo Flash.
Si se analiza el caso de la escritura en el sistema de archivos jffs2 con compresión zlib, se verifica que aunque se incrementa la latencia promedio de la función de compresión a 2195 microsegundos, el tiempo promedio total de la system call es de 5690. Es decir se logra una reducción significativa en el tiempo de escritura del dispositivo Flash.

Cuadro 1: Resultados de las pruebas

	write()	compress()	read()	decompress()
sin compresión	8240,17	1,44	1735,98	–
compresión zlib	5689,69	2194,88	2390,05	633,51

Sin embargo, deben destacarse dos hechos que se manifiestan en las pruebas y comprueban algunas afirmaciones que suelen aparecer en la literatura relacionada. Por un lado, se puede verificar que el algoritmo zlib es varias veces más rápido al descomprimir que al comprimir. compress() introdujo una latencia promedio de 2194,88 microsegundos por bloque mientras que decompress() sólo agregó 633,51 de tiempo promedio a la lectura de un bloque. Esta característica puede ser importante a la hora de elegir un sistema de archivos para un sistema embebido. Por otro lado, muchos benchmarks para sistemas de archivos suponen que todo lo que se lee del dispositivo debió ser escrito en algún momento, con lo cual plantean el tiempo combinado de escritura y lectura como

el indicador más importante. Por lo tanto, si se analiza la performance general, teniendo en cuenta el tiempo que consumen las operaciones de lectura y escritura en conjunto, se llega a la conclusión de que el uso de compresión incrementa la eficiencia del sistema de archivos y el dispositivo.

6 Conclusiones

El mundo de los sistemas embebidos ha venido creciendo de manera sostenida en los últimos años y, en este contexto, el futuro de la tecnología Flash es realmente alentador. Ya que se tiende a la ubicuidad de las computadoras y electrodomésticos inteligentes e integrados, se presenta la necesidad de contar con memorias pequeñas, baratas y con grandes capacidades de almacenamiento.

Además de sus otras ventajas, los sistemas de archivos estructurados como logs ofrecen una oportunidad en la cual la compresi´on de datos puede ser utilizada sin los problemas de asignación y fragmentación que se presentan en las aproximaciones más convencionales.

Utilizando una técnica de compresión suficientemente rápida, el tiempo utilizado en la compresión y descompresión se compensa por el tiempo ganado por la menor cantidad de accesos al dispositivo físico.

Claramente, esta afirmación está condicionada por la relación entre la velocidad de la CPU y la velocidad del controlador del dispositivo de almacenamiento. Sin embargo, ya que las velocidades de CPU están creciendo más rápido, en términos relativos, que las velocidades de los discos, se puede esperar que la compresión al vuelo disminuya la latencia y mejore el rendimiento de los sistemas de almacenamiento.

7 Referencias

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

[2] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. http://developer.intel.com/design/flcomp/applnots/297816.htm.

[3] Linux Devices. Linux Devices. http://www.linuxdevices.com.

[4] GNU. The GNU C Library Reference Manual. www.gnu.org.

[5] Intel. Using the RDTSC Instruction for Performance Monitoring. http://developer.intel.com.

[6] iozone. iozone. http://www.iozone.org.

[7] MIT. One Laptop Per Child. http://wiki.laptop.org.

[8] Inc Red Hat. eCos – Embedded Configurable Operating System. http://sources.redhat.com/ecos/.

[9] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems.

26

[10] Matthew Simpson. Analysis of Compression Algorithms for Program Data. University of Maryland.

[11] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The Case for Compressed Caching in Virtual Memory Systems. Dept. of Computer Sciences, University of Texas, Austin.

[12] David Woodhouse. The Journalling Flash File System. ACM Transactions on Computer Systems.

[13] Zlib. Informacion Tecnica sobre zlib. http://www.zlib.net.