Compresión transparente para sistemas embebidos

Matías Zabaljáuregui
matiasz@info.unlp.edu.ar

La Plata, Febrero de 2007

Indice General

1 Introducción

Las ventajas de las técnicas de compresión al vuelo 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.

Por ejemplo, la memoria disponible insuficiente en sistemas embebidos es un problema crítico que afecta la performance y confiabilidad de los dispositivos. Una posible solución para manejar este problema es comprimir bloques de memoria accedidos infrecuentemente cuando el sistema se queda sin memoria, y descomprimir la memoria cuando se necesite nuevamente, liberando memoria que puede ser realocada[11]. Por otro lado, la compresión adaptativa y transparente de los datos enviados por la red permite reducciones sustanciales del uso del ancho de banda disponible en los enlaces de comunicaciones.

Desde hace tiempo se plantea la opción de comprimir los datos que se almacenan en los discos o dispositivos similares como una forma de incrementar el espacio de almacenamiento efectivo. 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. Actualmente, la tecnología Flash es la mejor sustitución de un disco rígido en un dispositivo embebido. Tiene algunas características que la posicionan como la solución más conveniente: es rápida, consume menos energía y produce menos calor que un disco rígido, y no tiene partes móviles. A diferencia de un disco, el procesador puede acceder directamente a los bits almacenados en la memoria Flash como lo hace con una RAM o ROM, haciendo posible la ejecución en el lugar (eXecution In Place o XIP) reduciendo de esta forma los requerimientos de memoria RAM del dispositivo[1].

El sistema de archivos estudiado se basa en el concepto de estructurar los datos como un conjunto de entradas secuenciales en un log, en lugar de acceder aleatoriamente a los bloques de un disco. Más allá de las ventajas que presenta este diseño con respecto a la forma de acceso de las memorias Flash, los sistemas de archivos estructurados en logs facilitan varios aspectos de la implementación de la compresión al vuelo de los datos.

Para comprobar los beneficios en espacio de almacenamiento y medir el

[1] Esto sólo se aplica a algunas variantes de Flash.

overhead introducido por el procesamiento extra en compresión y descompresión, se escribieron programas de benchmark específicos para este trabajo y se agregó código al kernel Linux para calcular las latencias. Las pruebas se realizaron sobre hardware prototípico proveniente del proyecto One Laptop Per Child del MIT.

El resto del informe se organiza de la siguiente manera: la sección 2 introduce conceptualmente los algoritmos más utilizados para la compresión de datos, prestando mayor atención a cuestiones de eficiencia en el uso de recursos y complejidad temporal ya que, en este contexto, estos indicadores son tan importantes como la relación de compresión. La sección 3 presenta las ideas relacionadas con la compresión de datos incluida como una funcionalidad adicional de un sistema de archivos y explica los principios en los que se basan los sistemas de archivos estructrados como logs y el motivo por el cual facilitan la compresión de los datos. La sección 4 describe brevemente la tecnología Flash y las alternativas de software para manejarlas, y se mencionan las particularidades del Journaling Flash sistema de archivos, en particular con respecto a la compresión. Finalmente, la sección 5 presenta en detalle la metodología y los resultados de las pruebas realizadas sobre el dispositivo mencionado.

2 Algoritmos de compresión de datos

Hay algunas características deseables que debe tener un compresor de datos para ser efectivo en el contexto de los sistemas embebidos. Estas incluyen la habilidad de comprimir los datos con una relación suficiente para que se justifique el overhead, poco uso persistente de memoria y una gran velocidad de compresión y descompresión. Los indicadores de estas variables son escenciales al elegir el algoritmo de compresión más apropiado.

Existen dos categorías principales de técnicas de compresión de datos: aquellos que utilizan métodos estadísticos y aquellos que requieren el uso de un diccionario. Ambas técnicas son muy utilizadas pero los esquemas basados en diccionario tienden a ser más usados para aplicaciones de archivos, mientras que las situaciones de tiempo real típicamente requieren esquemas de compresión estadísticos. La causa de esto es que los métodos basados en diccionario tienden a ser lentos en la compresión y más veloces para descomprimir, mientras que los métodos estadísticos son igualmente veloces en ambas operaciones.

En el dominio estudiado, la compresión de los datos intenta realizarse con una aproximación de tiempo real (aunque no se exige la rigurosidad de los estándares de tiempo real hard). Sin embargo, como cada bloque de datos que se comprima eventualmente será descomprimido, la velocidad combinada de compresión y descompresión puede ser tan importante como las velocidades de cada una de estas operaciones por separado. Por lo

tanto, ambos tipos de algoritmos, e incluso combinaciones de estas técnicas deberían compararse para obtener el mejor resultado. La tabla 1 sumariza información relacionada con la performance esperada de cada algoritmo.

Cuadro 1: Algoritmos de Compresión de datos

Algoritmo	Tipo	Complejidad Temporal	Relacion de comp.	Memoria persis.
Huffman	Estadístico	N (n + log (2n-1)) + Sn	Promedio	Ninguna
Aritmético	Estadístico	N (log (n) + a) + S n	Promedio	Ninguna
PPM	Estadístico	dependiente del contexto	Buena	Ninguna
CTW	Estadístico	dependiente del contexto	Buena	Ninguna
LZO	Diccionario	aprox. N (d)	Buena	Ninguna
WKdm	Híbrido	aprox. N (d)	Buena	Ninguna
WKS	Híbrido	aprox. N (d)	Buena	Ninguna

2.1 Métodos Estadísticos

Los esquemas estadísticos de compresión determinan el código de salida basados en la probabilidad de ocurrencia de los símbolos de entrada y son típicamente utilizados en aplicaciones de tiempo real. Como los algoritmos de compresión y descompresión tienden a ser simétricos, la compresión y descompresión usualmente requiere la misma cantidad de tiempo para completarse.

2.1.1 Compresión Huffman

El método Huffman es tal vez la técnica más comúnmente utilizada de compresión estadística. Durante el proceso de codificación, éste método construye una lista de todos los símbolos de entrada, ordenados en base a sus probabilidades. El algoritmo luego construye un árbol, con un símbolo en cada hoja, y recorre el árbol para determinar los códigos para cada símbolo. Los símbolos con más probabilidad de ocurrencia tienen códigos más cortos. La decodificación utiliza el código para recorrer el árbol hasta llegar al símbolo.

La complejidad en el tiempo de una implementación adaptativa de la codificación Huffman es lineal: N (n + log (2n-1)) + Sn, donde N es el número total de símbolos de entrada, n es el número real de símbolos distintos y S es el tiempo requerido para rebalancear el árbol si fuera necesario.

2.1.2 Codificación Aritmética

Las implementaciones reales de la codificación aritmética son muy similares a las de codificación Huffman, aunque superan a estas últimas en la rela-

ción de compresión. El método Huffman asigna un número entero de bits a cada símbolo, mientras que la codificación aritmética asigna un único código extenso a la cadena de entrada completa. Por ejemplo, idealmente a un símbolo con probabilidad 0,4 se le debería asignar un código de 1.32 bits, pero será codificado con 2 bits usando el método Huffman. Es por esta razón que la codificación aritmética tiene el potencial de comprimir datos en el límite teórico.

La codificación aritmética combina un modelo estadístico con un paso de codificación que consiste en algunas operaciones aritméticas. El modelo más sencillo tendría una complejidad temporal lineal de N (log (n) + a) + S n, donde N es el número total de símbolos de entrada, n es el número real de símbolos distintos, a es la aritmética a ser realizada y S es el tiempo requerido, si se necesita, para mantener las estructuras de datos internas.

2.1.3 Modelos basados en el contexto

Prediction with Partial string Matching (PPM) es un modelo estadístico basado en el contexto, muy sofisticado, usado con codificadores aritméticos. La idea es asignar una probabilidad a un símbolo dependiendo no sólo de su frecuencia de ocurrencia, sino que también se toma en cuenta la forma en que aparece. PPM intenta matchear el contexto de orden más alto con con el símbolo actual. Si no se encuentra una coincidencia, el algoritmo busca un contexto de más bajo orden. Buscar los contextos mencionados puede ser costoso, especialmente si la entrada es muy desestructurada. La complejidad de tiempo depende de a través de cuantos contextos buscará el algoritmo.

Como los otros modelos estadísticos, *Context-Tree Weighting (CTW)* es un método para predecir la probabilidad de ocurrencia del próximo símbolo de entrada. El algoritmo examina una cadena de entrada dada y los d bits que la preceden, conocida como el contexto. Se construye un árbol de profundidad d donde cada nodo corresponde a un substring del contexto. Luego se examina el próximo bit de la entrada y el árbol es actualizado para contener a la nueva subcadena y usado para predecir la probabilidad de un contexto. Esta información luego es usada por un codificador aritmético para comprimir los datos. CTW, como PPM es capaz de alcanzar una relación de compresión alta, pero la complejidad temporal depende de la cantidad de contextos que tenga que analizar el algoritmo.

2.2 Métodos de diccionario

La compresión por diccionario no usa modelos de predicción estadística para determinar la probabilidad de ocurrencia de un símbolo particular, sino que almacena cadenas de símbolos de entrada en un diccionario. Estas técnicas son típicamente usadas en aplicaciones de archivo, como compress y gzip,

porque el proceso de decodificación tiende a ser más rápido que el de codificación.

2.2.1 Lempel-Ziv

La codificación Lempel-Ziv y sus muchas variaciones son, probablemente, los métodos más populares de compresión utilizados en aplicaciones de compresión de archivos. La variante más común, LZ77 o Compresión Sliding Window, hace uso de una ventana deslizante que consiste en un buffer de búsqueda, o diccionario, y un buffer no comprimido. Una cadena de símbolos es leida del buffer no comprimido, y se busca una coincidencia con la misma cadena en el buffer de búsqueda. Si se encuentra, se escribe en la salida un índice a la ubicación del string en el diccionario. Idealmente, el codificador selecciona la coincidencia más larga disponible en el buffer de búsqueda, aunque esto suele ser configurable ya que condiciona la performance del algoritmo, .

Ya que los métodos de compresión Lempel-Ziv requieren tiempo para buscar en el diccionario, la compresión es usualmente mucho más costosa que la descompresión. Muchas técnicas de compresión tienen sus raices en LZ77 y su sucesor LZ78. Por ejemplo, LZO (Lempel-Ziv-Oberhumer) es una implementación moderna de éste algoritmo que tiene una complejidad temporal lineal de aproximadamente N (d) donde N es el número total de símbolos de entrada y d es el tamaño del diccionario. Poseé algunas características interesantes y por lo tanto es muy utilizada para aplicaciones en entornos embebidos:

- Es un algoritmo sin pérdida
- La descompresión es simple, muy rápida y no requiere memoria RAM
- Permite distintos niveles de compresión
- Requiere como máximo 64 KB de memoria para la compresión, aunque en el nivel mínimo sólo utiliza 8 KB

2.3 Métodos Híbridos

Los métodos híbridos de compresión tienen características tanto de los métodos estadísticos como de los métodos basados en diccionario. Estos algoritmos usualmente incluyen un esquema de diccionario en los casos en que pueden asumirse cuestiones que simplifican los datos de entrada.

2.3.1 WKdm

Los algoritmos de compresión WKdm son un ejemplo de método híbrido diseñado para comprimir datos de manera eficiente y muy rápido. Fueron

propuestos en el trabajo *The case for compressed caching in virtual memory systems*[11] como los algoritmos a ser utilizados en la propuesta de memoria cache comprimida realizada por sus autores. Para detectar y aprovechar las regularidades de los datos, el compresor mantiene un diccionario de sólo 16 entradas y comprime cada palabra de entrada basado en ciertas condiciones que le permiten clasificarlas. La complejidad temporal de este algoritmo es similar a la de LZO, pero utiliza un diccionario de mucho menor tamaño.

WKS es una versión modificada de Wkdm. Esta forma del algoritmo requiere mucho menos memoria ya que soporta compresión y descompresión en el lugar (in-place), sin tener que copiar los datos a un buffer intermendio.

2.4 DEFLATE/zlib

En esta sección se explica el algoritmo utilizado para comprimir datos, de manera transparente, en el sistema de archivos estudiado en este trabajo. 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 con 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.

Algunas aplicaciones conocidas son:

- El kernel Linux, donde se utiliza para implementar protocolos de red comprimidos, sistemas de archivos comprimidos y para descomprimir la imagen misma del kernel en el momento de la inicialización.
- libpng, la implementación de referencia para el formato de imágenes PNG, la cual especifica a DEFLATE como el mecanismo de compresión de los datos de sus bitmaps.
- El servidor http Apache, el cual usa zlib para implementar la compresión de HTTP/1.1
- El cliente y servidor OpenSSH, el cual utiliza zlib para realizar la compresion del protocolo SSH.
- La librería de seguridad GnuTLS, que puede utilizar zlib para comprimir conexiones TLS, de manera opcional.

2.4.1 El algoritmo DEFLATE

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.

Como se mencionó anteriormente, LZ77 funciona encontrando secuencias de datos que están repetidas. En DEFLATE la ventana deslizante es de 32KB, es decir que el compresor y el descompresor tienen un registro de cuales fueron los últimos 32768 (32 * 1024) caracteres. Cuando la próxima secuencia de caracteres no comprimidos es idéntica a una que se encuentra en la ventana deslizante, se reemplaza por dos números: una distancia (entre 1 y 32768 bytes) que representa cuan lejos en la ventana deslizante comienza la secuencia y una longitud (entre 3 y 258 bytes) que indica el número de caracteres para los cuales la secuencia coincide.

La segunda etapa en el proceso de compresión consiste en la codificación Huffman del resultado del método LZ77. Sin embargo, 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 el método de Huffman clásico, el mismo conjunto de elementos y pesos podría generar múltiples árboles. En la variación utilizada por el estandar 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 reestricciones en la construcción del árbol de Huffman, existe un único árbol para cada conjunto de elementos y sus respectivas longitudes de código. Éstas 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. Hay tres modos de compresión disponibles:

- Datos sin comprimir. Esta opción es muy útil para datos que ya han sido comprimidos. Este modo expande los datos con algunos bytes, pero no genera tanto overhead como si se intentara aplicar alguno de los otros modos de compresión.

- Compresión, con LZ77 primero y luego con codificación Huffman. Los árboles utilizados para comprimir en este modo están definidos en la especificación de DEFLATE, por lo que no se requiere espacio extra para almacenarlos.
- Compresión, utilizando LZ77 y luego la codificación Huffman con la longitud de los códigos (para reconstruir los árboles) que el compresor almacena junto con los datos.

Los datos son divididos en bloques y cada uno de estos bloques usa un único modo de compresión. Si el compresor desea cambiar de un modo a otro debe terminar el bloque actual y comenzar uno nuevo.

2.4.2 La implementación zlib

Zlib es la implementación más utilizada de DEFLATE, y presenta algunas características interesantes para su uso en diversos entornos. Durante la compresión, el compresor debe conocer la cantidad máxima de tiempo utilizado en buscar secuencias de caracteres que coincidan. La implementación de referencia zlib permite al usuario seleccionar un nivel de una escala de compromiso entre la relación de compresión y la velocidad de codificación. Las opciones van desde 0 (no intenta comprimir) hasta 9, que representa la máxima capacidad de compresión de zlib.

La implementación en el Kernel Linux (al igual que la implementación de referencia) incluye dos variables, *windowBits* y *memLevel*, que permiten modificar el comportamiento por defecto. Estos parámetros pueden ser usados para ajustar la memoria y el tiempo consumidos por el compresor y descompresor (por ejemplo para entornos con hardware limitado), aunque influyen en la relación de compresión lograda por el algoritmo.

A través de pruebas empíricas sobre la implementación de referencia, se comprobó que el método DEFLATE es capáz de lograr relaciones de compresión de 1000:1. El caso de prueba es un archivo de 50MB de bytes en cero, que se comprimió a sólo 49KB. Sin embargo, debe notarse que estos niveles de compresión son excepcionales y debe suponerse que las relaciones típicas, dependiendo del tipo de datos, están entre 2:1 y 5:1.

3 Introducción a 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 por lo que incorporar técnicas de compresión en un sistema de archivos ofrece la ventaja de tener mayor espacio efectivo de almacenamiento. 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 mas lentos en una computadora).

3.1 Objetivos de diseño de un sistema de archivos comprimido

Existe 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:

- El sistema de archivos comprimido debe diseñarse con el objetivo fundamental de incorporar la característica de compresión y descompresión al vuelo de los datos mientras se mantiene la performance comparable con los otros sistemas de archivos.
- También se intenta que estas características sean transparentes para el usuario, el cual debería ignorar el hecho de que los archivos son almacenados de forma comprimida en el disco. El tamaño y otros atributos del archivo visibles para el usuario, deberían permanecer igual que los del archivo sin comprimir.
- Además se busca proveer compatibilidad binaria con los programas existentes. Todos los programas deberían trabajar sin modificación o recompilación.
- Los datos en un sistema de archivos son almacenados en el disco utilizando bloques lógicos[2]. Se puede evaluar si es conveniente hacer la compresión de datos a nivel de archivo o al nivel de bloque lógico. 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,

[2]Los bloques lógicos son abstracciones de los bloques físicos del disco, provistas por el sistema operativo. Típicamente el tamaño del bloque logico es tomado como un múltiplo fijo del tamaño del bloque físico.

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 del kernel ya que aunque el tamaño de bloque lógico siempre es un multiplo entero del del bloque físico, el número de bloques físicos requeridos para almacenar un bloque lógico depende de cuanto sea comprimido ese bloque de datos. Por ejemplo, un bloque lógico de 4K puede ocupar uno, dos, tres o cuatro bloques físicos de 1K, dependiendo del nivel de compresión logrado. Por lo tanto, el kernel debe soportar tamaños variables de bloques logicos.

- 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 compreión 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. Además, ya que los directorios suelen ser archivos de tamaño pequeño, y más frecuentemente accedidos, se puede elegir no comprimirlos.

3.2 Compresión en un sistema de archivos estructurado como log

Como se mencionó en la sección anterior, resulta interesante la idea de construir un sistema de archivos que comprima los datos que se almacenan en el disco. En primer lugar, entrarán más datos en el disco y luego, si se coloca un compresor rápido en el camino de los datos al disco, se incrementará (en el orden de la relación de compresión) la velocidad de transferencia efectiva al disco y por lo tanto la performance del sistema. Sin embargo, 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 proveén una compresión uniforme de todos los datos. Cuando un bloque de un archivo es sobreescrito, los nuevos datos pueden ser comprimidos con otra relación con respecto a los datos anteriores, por lo tanto, el sistema no podrá simplemente sobreescribir 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. El cuadro 2 muestra la variación en la relación de compresión cuando se incrementa el tamaño de bloque, con un simple algoritmo adaptativo de compresión[10]. Los detalles varían para diferentes algoritmos y diferentes datos, pero la tendencia suele ser la misma: Los bloques más grandes logran mejor relación de compresión.

Cuadro 2: Mejora en la relación de compresión con bloques más grandes

Tamaño bloque de entrada (bytes)	Relación de compresión
1K	68 %
2K	63 %
4K	59 %
8K	55 %
16K	53 %
32K	51 %

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 sistema 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 sobreescriben bloques, no es necesario realizar ac-

ciones especiales cuando los nuevos datos se comprimen con otra relación y, como los bloques son escritos secuencialmente en el disco, se elimina la fragmentación.

3.2.1 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 la file cache y luego escribiendo todos los cambios secuencialmente en una única operacion 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[3]. 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, éstos 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 via 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

[3]Representar un sistema de archivos como un único árbol no es una idea exclusiva de los sistemas de archivo estructurados como log. Reiserfs y TUX2 trabajan de esta forma.

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

4 La tecnología Flash y JFFS2

4.1 Memorias Flash

Flash es una tecnología de almacenamiento cuya utilización está creciendo notablemente en sistemas embebidos porque proveé almacenamiento basado en estado sólido con alta densidad y confiabilidad, a un costo relativamente bajo.

Las memorias Flash son de tipo no volátil, es decir que la información que almacena no se pierde en cuanto se desconecta de la corriente, 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.

Las capacidades de almacenamiento de las tarjetas que integran memorias Flash comenzaron en 8 MB pero actualmente se pueden encontrar en el mercado tarjetas de hasta 32 GB (anunciadas en setiembre del 2006 por la empresa Samsung Electronics). El costo de estas memorias es muy bajo respecto a otro tipo de memorias similares y ofrece rendimientos y características muy superiores.

Flash es una evolución de las memorias EEPROM (Electrically Erasable Read Only Memory) y se dispone de dos variantes principales: Flash NOR y Flash NAND. Ambos tipos de memoria tienen sus celdas seteadas en un 1 lógico cuando están borradas y las ponen a 0 en una operación de escritura. Los chips Flash se componen de bloques que suelen ser de 128KB en la variante NOR y de 8KB en las Flash NAND. El borrado, o el reseteo de cero a uno, no puede ser realizado en bits individuales sino que debe realizarse por bloques y el tiempo de vida de un chip Flash se mide en ciclos de borrado, siendo normalmente de 100.000 ciclos por bloque, aproximadamente. Para asegurar que ningún bloque alcanza su límite antes que el resto, el software suele intentar que los ciclos de borrado sean distribuidos homogéneamente por todo el chip. Éste proceso es conocido como *wear levelling.*

La densidad de almacenamiento de los chips es bastante mayor en las memorias NAND aunque su costo sea menor que el de las memorias NOR, y esto se debe a ciertas características que las distinguen. El acceso NOR es aleatorio para lectura y orientado a bloques para su modificación. Sin embargo, NAND ofrece tan solo acceso directo para los bloques y lectura secuencial dentro de los mismos. En la escritura de NOR podemos llegar a modificar un solo bit. Esto destaca la limitada reprogramación de las

NAND que deben modificar bloques o palabras completas. La velocidad de lectura es muy superior en NOR (50-100 nanosegundos) frente a NAND (10 microsegundos de la búsqueda de la página + 50 nanosegundos por byte). La velocidad de escritura para NOR es de 5 microsegundos por byte frente a 200 microsegundos por página en NAND. La velocidad de borrado para NOR es de 1 segundo por bloque de 64 KB frente a los 2 milisegundos por bloque de 16 KB en NAND. La fiabilidad de los dispositivos basados en NOR es realmente muy alta, ya que es relativamente inmune a la corrupción de datos y no tiene bloques erróneos. En cambio, los sistemas NAND requieren software de corrección de datos y existe la posibilidad de que queden bloques marcados como erróneos e inservibles. En resumen, los sistemas basados en NAND son más baratos y densos, pero carecen de fiabilidad por hardware, lo que demuestra la necesidad de un sistema de archivos robusto.

4.1.1 Capas de traducción y sistemas de archivos para memorias Flash

Hasta hace poco tiempo, la mayoría del software de almacenamiento de archivos en memoria Flash emulaba un dispositivo orientado a bloques (DOB) [4] con sectores de 512 bytes y luego se utilizaba un sistema de archivos estándar sobre el DOB emulado.

El método más sencillo para lograr esto es usar un mapeo uno a uno desde el DOB emulado al chip Flash. En las escrituras, se simulan los sectores de 512 bytes leyendo un bloque completo de la Flash a un buffer en memoria, modificando el buffer apropiadamente y finalmente borrando y reescribiendo el bloque completo. Esta aproximación no proveé wear levelling y es extremadamente insegura por la posible pérdida de alimentación eléctrica entre el borrado y la reescritura de los datos (los dispositivos embebidos suelen ser inseguros en este sentido). El driver de Linux *mtdblock* proveé esta funcionalidad con algunas optimizaciones para prevenir una cantidad excesiva de ciclos de borrado.

Para emular un dispositivo por bloques de manera apropiada para ser utilizado por un sistema de archivos estándar, se requiere una solución más sofisticada. Por ejemplo, para proveer wear levelling, los sectores del DOB emulado se pueden almacenar en direcciones variantes en el chip Flash y se puede utilizar una capa de traducción para mantener un registro de la ubicación actual de cada sector del DOB emulado (en realidad, esta capa de traducción termina siendo una forma de sistema de archivos con journaling).

La capa de traducción más utilizada es una componente de la especificación PCMCIA llamada *Flash Translation Layer*[2]. Más recientemente, una variante diseñada para ser utilizada con Flash NAND, conocida como NFTL, se ha vuelto popular en los dispositivos DiskOnChip[3].

[4] Es decir, un dispositivo de acceso aleatorio y cuya unidad de lectura y escritura son bloques en lugar de bytes. Por ejemplo, un disco rígido.

Desafortunadamente, tanto FTL como la nueva NFTL están protegidas por patentes. Linux soporta ambas capas de traducción, pero se consideran tecnologías obsoletas y se mantienen sólo por compatibilidad. Además del problema de las patentes, el hecho de usar una forma de sistema de archivos con journaling para emular un DOB sobre el cual se utilizará un sistema de archivos estándar con journaling resulta demasiado ineficiente. Una forma más razonable de utilizar la tecnología Flash es a través de un sistema de archivos diseñado específicamente para ser usado con tales dispositivos, sin capas extras de traducción.

Diseñar un sistema de archivos eficiente para las memorias flash se ha convertido en una carrera vertiginosa y compleja, ya que, aunque NOR y NAND son tipos de memoria Flash, tienen características muy diferentes entre sí a la hora de acceder a los datos. Los dos sistemas de archivos que se disputan el liderazgo para la organización interna de las memorias Flash son JFFS (Journaling Flash File System) y YAFFS (Yet Another Flash File System). YAFFS fue gestado como un sistema de archivos diseñado para memorias Flash del tipo NAND y tienen en cuenta, según sus desarrolladores, las características específicas de esta variante. JFFS, en su primer versión, fue pensado para memorias NOR. Posteriormente se reescribió gran parte de su código y se creó JFFS2, que se adaptó a la tecnología NAND además de agregar otras características interesantes como la compresión de los datos.

4.2 Journaling Flash File System

Los objetivos del diseño de JFFS[12] 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. Los nodos (entradas en el log) que contienen datos y metadatos son almacenados en el chip de la Flash secuencialmente, progresando de manera estrictamente lineal a lo largo del espacio de almacenamiento disponible. Cada nodo está asociado a un único inodo y comienza con un encabezado que contiene el número de inodo, los metadatos asociados a ese inodo, y puede contener una cantidad variable de datos del archivo correspondiente, para lo cual deberá incluir el desplazamiento en el archivo a donde los datos pertenecen.

Existe un orden total entre todos los nodos que pertenecen a un inodo individual, y se mantiene a través de números de versiones crecientes en cada nodo correspondiente a un mismo inodo. Cuando un nodo contiene datos o metadatos de un inodo que se solapan con los de un nodo anterior, se dice que el nodo anterior es obsoleto. El espacio ocupado por nodos obsoletos

es llamado *dirty space*. El borrado de un inodo se realiza marcando un flag en los metadatos de un inodo y cuando se cierra la última referencia de un proceso al archivo borrado, se marcan los nodos como obsoletos.

JFFS escribe nodos en el log hasta que el dispositivo se queda sin espacio disponible. En ese momento, el sistema necesita reclamar el dirty space, que contiene nodos marcados como obsoletos. Esta tarea se suele denominar como garbage collection y se implementa como un thread del kernel que se ejecuta en background en Linux.

En el momento de montar el sistema de archivos, se lee todo el dispositivo para interpretar cada nodo y reconstruir la jerarquía de directorios completa y un mapa para cada inodo que indica la ubicación física de los bloques de datos. JFFS suele ser criticado porque la operación de montaje puede ser algo costosa en tiempo. Esto se debe a que durante esta operación, se realizan un escaneo físico del medio y lecturas adicionales para armar un mapa del sistema de archivos en memoria y poder acceder rápidamente a los datos en las operaciones de lectura y escritura posteriores. El sistema de archivos JFFS requiere cierta cantidad de espacio disponible entre el principio y el final del log para asegurar que es posible realizar el procedimiento de garbage collection.

Los cambios en los metadatos (tales como el propietario o los permisos sobre un archivo) se realizan simplemente escribiendo un nuevo nodo al final del log. Las escrituras sobre un archivo son similares, sólo que el nodo escrito tendrá los datos asociados con la transferencia.

4.2.1 La evolución de JFFS

La versión original fue utilizada en sistemas embebidos de la empresa Axis, con el kernel Linux 2.0. Luego que el código fue liberado, fue portado al kernel de desarrollo 2.3 y posteriormente Red Hat lo incluyó en su kernel y comenzó a ofrecer soporte comercial para sus clientes.

Sin embargo, surgieron algunos reclamos por parte de los clientes, entre los más importantes la falta de compresión de los datos. Por el costo de los chips Flash y el intento constante de compactar mayor funcionalidad en los dispositivos embebidos, la compresión era un requerimiento importante de gran parte de los usarios potenciales de JFFS. Además se volcó un esfuerzo considerable para lograr un diseño portable del sistema de archivos. En particular, fue incluido en eCos[8], el sistema operativo para dispositivos embebidos de Red Hat.

Los datos incluidos en los nodos pueden ser comprimidos usando alguno de los algoritmos de compresión que pueden ser agregados como pluggins al código de JFFS2. Cada nodo puede comprimirse con un algoritmo distinto o ser del tipo *none* (que significa que los datos se almacenan sin comprimir) o del tipo *zero* (que indica que los datos son todos bytes en cero). Aunque se desarrollaron dos algoritmos de compresión específicamente para ser usados

en JFFS2 (rtime y rubin), zlib es la opción por defecto.

Como en otros sistemas de archivos con compresión de datos, se aprovecha el paso intermedio que ofrece la page cache para comprimir y descomprimir los datos que se escriben o se leen del disco. En el caso de la escritura, primero se copia el buffer de datos en espacio de usuario a una página en la page cache (por defecto, una página es de 4096 bytes en la plataforma x86). Más adelante en el tiempo (o inmediatamente, si la operación se configuró como sincrónica), se ejecuta el método *commit_write()* sobre esa página, el cual invoca a la función de compresión y luego escribe los datos al dispositivo.

En una lectura, los datos pueden estar en la page cache si han sido accedidos recientemente. Caso contrario, el kernel aloca una página en la page cache e invoca al método *readpage()*, cuya implementación en JFFS2 lee los datos del dispositivo y, sólo si fuera necesario, invoca a la función de descompresión. Finalmente, los datos descomprimidos son colocados en la page cache, por lo que los accesos posteriores no requieren descomprimirlos nuevamente (al menos mientras éstos se mantengan válidos en la page cache). Para facilitar una rápida descompresión de los datos durante un requerimiento readpage(), los nodos no contienen más que una única página de datos, de acuerdo con el tamaño de página del hardware. Esto significa que las imágenes del sistema de archivos JFFS2 no son portables entre ciertas máquinas.

La operación de JFFS2 es muy similar a la de JFFS: se escriben nodos hasta llenar un bloque de borrado de la flash y luego se elige otro bloque, desde una lista de bloques libres, para seguir escribiendo datos. Cuando la lista de bloques libres alcanza una longitud mínima definida, se inicia el thread de garbage collection para mover nodos dede un bloque antiguo a un bloque nuevo hasta que pueda reclamarse el espacio del bloque antiguo, realizando una operacion de borrado de bloque de la memoria flash y agregándolo a la lista de bloques libres.

Finalmente, existe una característica mencionada previamente que es la de ejecución en el lugar (XIP o eXecute InPlace) y que suele incrementar la eficiencia del dispositivo. Cuando se ejecuta un binario almacenado en JFFS2, el código ejecutable es copiado de la memoria Flash a la RAM antes de que la CPU pueda ejecutarlo. Sin embargo, algunos sistemas de archivos permiten la ejecución desde el mismo chip Flash, evitando la copia extra a la memoria principal. En JFFS2, se ha decidido no incluir esta funcionalidad por varios motivos. Para este trabajo, el más importante es el hecho de que XIP y la compresión son mutuamente excluyentes, ya que si los datos están comprimidos no pueden ser utilizados directamente. Esta funcionalidad aún es motivo de discusión entre los desarrolladores de dispositivos embebidos[5].

[5]Supongamos una plataforma embebida prototípica con cantidades suficientes tanto de RAM como de Flash como para no necesitar ni compresión ni XIP. Si se desea disminuir el

JFFS2 se utiliza en un gran número de sistemas embebidos, como por ejemplo la distribución oficial de la computadora handheld iPAQ de Compaq, en la cual reemplazó al sistema de archivos de sólo lectura CRAMFS. Además, Red Hat ofrece soporte comercial para JFFS2 para los clientes que deseen usarlo en sus sistemas en producción. El proyecto OLPC, que aún se encuenta en un estado beta, se decidió originalmente por este sistema de archivos para sus notebooks de 100 dólares, sin embargo, aun se continúa estudiando las ventajas y desventajas de JFFS2 con respecto a otras alternativas de sistema de archivos para sistemas embebidos, como YAFFS.

5 Las pruebas realizadas con OLPC

En esta sección se describen las pruebas de performance 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.

5.1 Especificaciones de hardware de la placa OLPC

A continuación se describen los componentes de hardware que conforman los prototipos del MIT conocidos como placas alpha o placas A:

- CPU: AMD Geode GX-500@1.0W, 366Mhz
- Compatibilidad: X86/X87
- Chipset: South Bridge AMD CS5536
- Controlador de gráficos: Integrado con CPU Geode
- Memoria DRAM: 128MB, Dual DDR266
- BIOS: 1024 KB ROM flash con interfaz SPI; BIOS open-source LinuxBIOS

costo del hardware, puede ser necesario tomar una decisión entre reducir la RAM y utilizar XIP por un lado, o reducir la Flash y usar compresión de datos. Ya que, por ahora, los costos de la memoria Flash son sustancialmente mayores a los de la memoria RAM, la última opción será la más conveniente.

- Almacenamiento: Controlador SCL NAND Flash 512MB

5.2 La configuración del framework de compresión y el método utilizado para medir el tiempo

5.2.1 La compresión utilizada

Con un procesador de 366Mhz, y 512 MB de memoria Flash para el almacenamiento de datos, la OLPC ofrece las características de un dispositivo embebido de altas prestaciones. Sin embargo, la compresión de los datos se presenta como una solución viable para superar la limitación de espacio que presentan los escasos 512MB, más aún teniendo en cuenta que esta plataforma será utilizada como una PC y no como un dispositivo específico.

Es posible agregar pluggins (alogritmos de compresión) al framework de compresión ofrecido por JFFS2. La versión del sistema de archivos que se incluye en el kernel estándar ofrece un conjunto de algoritmos por defecto, aunque zlib es el que logró una mejor relación de compresión en todas las pruebas realizadas. Una vez definida la lista de algoritmos, es posible indicar una heurística de compresión que permite especificar el criterio por el cual se seleccionará automáticamente uno de los algoritmos incluidos en el conjunto. Por ejemplo, la heurística *size* probará cada uno de los algoritmos y elegirá aquel que logre la mejor relación de compresión para los datos.

Para las pruebas se utilizó la heurística *priority*, la cual permite asignar prioridades a los algoritmos e intenta comprimir con cada uno de ellos, en el orden definido por las prioridades, hasta que alguno tenga éxito en la operación. En este caso, se considera que la operación es exitosa cuando se logran comprimir los datos. La operacion no es exitosa cuando, al aplicar el algoritmo, el flujo de datos de salida tiene mayor longitud que el flujo de entrada.

5.2.2 Midiendo el tiempo para leer o escribir un bloque de datos

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 compresión 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. Como se estudió previamente, zlib utiliza una combinación de LZ77 con Huffman. Con

un archivo generado a partir de un único patrón, un algoritmo basado en diccionario será mucho más efectivo que el caso promedio.

Por lo tanto, para las pruebas realizadas para este trabajo, se escribieron dos programas específicos que miden el tiempo consumido por cada operación de bloque escrito o leido. Es decir cada resultado impreso por el programa representa el lapso de tiempo, en microsegundos, desde que se invoca a la system call correspondiente hasta que el kernel devuelve el control al proceso de usuario. Se decidió utilizar bloques de 4096 bytes, ya que es la unidad de compresión utilizada por jffs2 en esta plataforma. Estos programas permiten utilizar cualquier archivo para realizar las mediciones de tiempo, por lo que se gana la posibilidad de optar por distintos tipos de datos para estudiar el comportamiento de cada algoritmo de compresión.

En el caso de las escrituras, es necesario realizar operaciones sincrónicas[6] para evitar los beneficios de los accesos diferidos al dispositivo de almacenamiento que ofrece la page cache del kernel Linux. De esta forma cada invocación a *write()* devuelve el control al proceso de usuario sólamente cuando los datos hayan sido escritos efectivamente en la memoria Flash y, por lo tanto, el tiempo medido incluye la latencia en el acceso y el tiempo de transferencia de los datos al medio físico.

En el caso de las lecturas, es necesario desmontar y volver a montar el sistema de archivos para asegurar que la page cache no contiene páginas de datos pertenecientes al sistema de archivos estudiado antes de iniciar la lectura del archivo y, por lo tanto, el acceso obligado al dispositivo Flash en lugar de sólo a la RAM. Por otro lado, en las pruebas de lectura se puede observar el funcionamiento de la característica conocida como read-ahead del Virtual File System de Linux[1]. Cuando el kernel reconoce que se está realizando una lectura secuencial de un archivo, aprovecha los accesos al dispositivo físico para recuperar más datos que los solicitados por el usuario, de manera tal que en los siguientes requerimientos los datos ya estarán en memoria. Esto se puede observar claramente en el patrón de latencias de los accesos de lectura medidos en las pruebas. Desafortunadamente, el driver del dispositivo Flash no soporta reconfigurar el comportamiento de read-ahead sin modificar el código fuente, con lo cual se decidió realizar las pruebas en estas condiciones y luego utilizar los valores promedios.

En estos programas se hace uso de la función *gettimeofday()* y de estructuras de datos de tipo *struct timeval* que ofrecen resoluciones en el orden de los microsegundos. Como indica la documentación de la libc de GNU[4], es necesario utilizar estas funciones ya que para las pruebas es de interés conocer el tiempo que el proceso permanece bloqueado mientras se accede al dispositivo de almacenamiento. Las funciones *times()* o *clock()*, por otro lado, permiten conocer el tiempo que un proceso dado ocupa la CPU, pero no computan el tiempo de E/S. Se adjunta el código fuente de ambos

[6] Esto se logra utilizando el flag *O_FSYNC* en la llamada a *open()*

programas, llamados *zbwrite.c* y *zbread.c*.

5.2.3 Midiendo la latencia de comprimir o descomprimir los datos

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. Por ejemplo, el valor conocido como *jiffies* se incrementa con cada interrupción elevada por el timer de la máquina. Sin embargo, el kernel Linux 2.6, en arquitecturas x86, suele configurar el timer para que interrumpa al procesador con una frecuencia de 1000 HZ, con lo cual se obtiene una resolución en el orden de los milisegundos.

Afortunadamente, los fabricantes de CPU introdujeron una forma de contar ciclos del reloj de la CPU como un mecanismo sencillo y confiable para medir intervalos de tiempo. La mayoría de los procesadores modernos incluyen un registro contador que es incrementado en cada ciclo de CPU. Este contador es la única forma en que se pueden llevar a cabo tareas que requieren una gran resolución en la medición del tiempo. TSC (TimeStamp Counter), introducido en los procesadores x86 a partir del Pentium y presente en todas los diseños de CPU desde entonces, es un registro de 64 bits que cuenta ciclos del reloj de la CPU y puede ser leido tanto desde el kernel como desde el espacio de usuario. Desde código del kernel, existe un grupo de macros que permiten leer valores de 32 y 64 bits desde el TSC. En las pruebas realizadas se utilizó la macro *rdtscl()* que obtiene la parte baja del registro y la almacena en una variable de 32 bits. Finalmente, es necesario realizar el cálculo del tiempo, para expresarlo en microsegundos, teniendo en cuenta la frecuencia de la CPU (esto puede conocerse por las especificaciones del hardware y puede corroborarse con la información que el kernel exporta en el archivo /proc/cpuinfo). En el caso de la versión alpha del prototipo de la OLPC, el procesador es de 366MHz.

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()*. Se puede observar en los archivos read.c y write.c del directorio /linux-2.6.18.2/fs/jffs2 de las fuentes del kernel adjuntadas con el trabajo, que se agregó el código que mide los ciclos consumidos por las funciones virtuales[7] *jff2_decompress()* y *jffs2_compress()*. Como se indica en la documentación de Intel sobre el registro TSC[5], fue necesario utilizar la función *mb()* que implementa una barrera de memoria para evitar reordenamientos de instrucciones por parte del compilador que pueden interferir en la medición.

[7]Son los punteros a funciones que se setean con las direcciones de las funciones que implementan los algoritmos de compresión elegidos en la configuración.

A continuación se muestran los extractos de código perteneciente a read.c y write.c:

read.c

```
...
mb();
rdtscl(x);
jffs2_decompress();
mb();
rdtscl(y);
printk(KERN_DEBUG "ciclos ocurridos durante decompress: \%li", y - x);
...
```

write.c

```
...
mb();
rdtscl(x);
jffs2_compress();
mb();
rdtscl(y);
printk(KERN_DEBUG "ciclos ocurridos durante compress: \%li", y - x);
...
```

5.3 Los resultados obtenidos

En el archivo llamado *resumen.xls* se puede observar el resumen de los datos recogidos durante las pruebas realizadas. El primer conjunto de datos corresponde a la configuración sin compresión de jffs2. El segundo conjunto representa las mediciones utilizando la compresión por defecto de jffs2, zlib. Para cada conjunto, hay tres columas de datos (excepto en la lectura sin compresión, ya que no se invoca a la función de descompresión en el caso de leer datos no comprimidos) que representan respectivamente:

- Escritura/Lectura de un bloque medida en microsegundos. Este dato se obtiene con los programas mencionados previamente y representa el tiempo consumido desde que se invoca a la system call correspondiente hasta que el kernel devuelve el control al usuario.
- Ciclos de CPU consumidos en comprimir o descomprimir los datos. Este dato se obtiene usando la macro *rdtscl()* en el código del kernel.
- Microsegundos consumidos en comprimir o descomprimir. Este valor se obtiene a partir de conocer los ciclos de CPU consumidos y la frecuencia del procesador.

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 la tabla, 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 3: 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

Las memorias Flash del tipo NAND suelen ser más lentas para escribir que para leer. En particular, la versión alpha del prototipo de OLPC tiene un controlador de Flash embebido en la CPU AMD Geode que tiene una tasa de transferencia muy baja[8]. Por lo tanto, el hecho de reducir a la mitad la cantidad de datos a ser transferidos a la memoria Flash parece ser conveniente más alla de que se inviertan más de 2000 microsegundos en esta tarea. Es decir, aunque se agrega una latencia considerable en la función de compresión, se disminuye el tiempo neto de escribir un bloque en la memoria Flash. Por lo tanto, la compresión de datos no sólo incrementa en un 100 % el tamaño efectivo de almacenamiento, sino que además incrementa la tasa de transferencia efectiva entre el dispositivo y la memoria ram, en el caso de la escritura de datos.
Este fenómeno no se presenta en el caso de las lecturas. Como se mencionó anteriormente, la lectura es una operación relativamente rápida

[8]En las lista de correo de los desarrolladores de la OLPC se mencionaba un máximo de 2.71 MB por segundo. Esto se solucionó en la versión beta del hardware agregando un controlador Flash diseñado en el MIT específicamente para la OLPC.

en estos dispositivos. Se puede apreciar la diferencia de tiempo entre realizar una llamada a write() y una llamada a read() sin compresión de datos: 8240 microsegundos versus 1736 microsegundos. Esta diferencia notable tiene que tener su origen en el hardware, ya que el procesamiento de la system call read() es similar al de write(). A diferencia de compress() que se invoca siempre, aunque los datos no vayan a comprimirse, decompress() sólo es invocada por el código de jffs2 cuando se encuentra un nodo comprimido. Por este motivo, en el caso de la lectura de datos sin comprimir, no existen datos de latencia para decompress. Por otro lado, el tiempo promedio para leer un bloque de datos comprimido es de 2390 microsegundos, mientras que la latencia introducida por la descompresión de los datos es de 633 microsegundos. Claramente, en el caso de la lectura el tiempo invertido en descomprimir no se compensa con la menor cantidad de datos leidos desde el dispositivo físico: La latencia promedio de lectura se incrementa en un 30 % aproximadamente.
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 (en la actualidad se están fabricando dispositivos con memorias Flash NAND de 32 Gb con un tamaño similar al de un disco duro de 2.5 pulgadas).
En este trabajo se presentaron las ventajas que ofrecen los sistemas de archivos con compresión al vuelo y transparente para los datos

almacenados en este tipo de memorias. Además de sus otras ventajas, los sistemas de archivos estructurados como logs ofrecen una oportunidad en la cual la compresión de datos puede ser utilizada sin los problemas de alocación y fragmentación que se presentan en las aproximaciones más convencionales.
Gracias a las pruebas realizadas sobre el hardware del proyecto OLPC, se ha comprobado la posibilidad de reducir el uso efectivo del espacio de disco gracias al uso de un algoritmo de compresión rápido y con una relación de compresión de 2:1, como lo es zlib, integrado en un sistema de archivos estructurado en logs, como JFFS2.
Además, 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.

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.

[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.