Explotando Localidad en Códigos basados en Punteros

E. Gutiérrez, S. Romero, O. Plata, E.L. Zapata

Resumen—

El aprovechamiento de la localidad es, en general, un aspecto crucial en la optimización de códigos, dada la creciente disparidad entre las velocidades de trabajo de procesadores y memoria. Aunque en el ámbito de códigos cuyas estructuras de datos son densas y basadas en *arrays* existen potentes técnicas de compilación orientadas a mejorar y explotar localidad, en el caso de los códigos basados en punteros, la situación no es tan favorable.

El desconocimiento a priori tanto de la forma de las estructuras de datos basadas en punteros como de las posiciones concretas donde se ubicará en tiempo de ejecución, convierten la optimización de este tipo de códigos, en un arduo reto.

En este trabajo se presentan algunas ideas, de cómo el análisis de forma de las estructuras basadas en punteros puede ayudar en la explotación de localidad de este tipo de códigos.

I. Introducción

El principio de localidad es una propiedad, por todos conocida, que presumimos que va a estar presente en la ejecución de nuestros programas. Precisamente suponer que esta propiedad es cierta es la base del diseño de una jerarquía de memoria que en los computadores convencionales surge como una solución que intenta paliar la diferencia (*gap*) de velocidades existentes entre el procesador y la memoria.

Por este motivo la explotación de localidad es un tema crucial cuando el objetivo es obtener una buena eficiencia en la ejecución de un código. En el caso de problemas regulares, los accesos se realizan a estructuras de datos estáticas, como arrays, y además vienen definidos por patrones que son conocidos en tiempo de compilación. En estos casos, el compilador puede detectar dependencias y encontrar reordenamientos frecuentemente, tanto de los datos como de la computaciones, sin cambiar con ellos el comportamiento del código.

Un caso de mayor complejidad lo constituyen los denominados códigos irregulares. Estos códigos aunque trabajan sobre estrucucturas estáticas como arrays, se caracterizan por accesos a través de indirecciones. Las referencias indirectas son desconocidas en tiempo de compilación, determinándose sus valores en la ejecución y además son causa de dependencias acarreadas por el lazo. Ello dificulta tanto el análisis de dependencias como el encontrar una forma efectiva de explotar localidad.

Departamento de Arquitectura de Computadores
Universidad de Málaga.
E-mail: {eladio,sromero,oscar,ezapata}@ac.uma.es.

Este trabajo ha sido financiado por el proyecto TIC2003-06623 del ministerio de Educación y Ciencia (CICYT).

No obstante se han realizado trabajos [1], [3], [2], [4] en ciertos patrones con indirecciones que se encuentran comúnmente en códigos científicos como son los asociados a operaciones conmutativas y asociativas denominadas reducciones. Estos métodos demuestran que aún desconociendo los valores que toman las indirecciones durante la compilación es posible encontrar transformaciones automatizables que aprovechen la localidad subyacente en los accesos.

Los códigos que hacen uso de estructuras basadas en punteros suponen un reto aún mayor desde el punto de vista de la localidad. Las estructuras empleadas por estos códigos se construyen y modifican durante la ejecución. Frente a los códigos regulares, no es posible a priori determinar cuál es el patrón de acceso y ni siquiera se conoce la estructura sobre la que se opera. Esta estructura vendrá definida por el grafo que forman los punteros y que se va creando en tiempo de ejecución.

Además los elementos de los que consta la estructura se van reservando dinámicamente a medida que se necesitan por lo que pueden estar dispersos en el espacio de memoria. Esta dispersión será mayor o menor según la política de alojamiento de nuevas posiciones y ésta no está en general bajo el control del programador. Esto evita además la posibilidad de hacer suposiciones acerca del lugar se encuentran, ya que ejecuciones diferentes del mismo programa con los mismos datos de entrada puede dar lugar a patrones de accesos diferentes.

A título ilustrativo se muestran en la tabla I las características de localidad de un código ejemplo, implementado en una versión basada en arrays y otra basada en punteros. Estas características de localidad se han expresado como fallos cache en dos niveles de la jerarquía de memoria (L1 y L2). Básicamente el programa recorre una estructura, bien vectorial, bien lista enlazada, donde se visita una lista de componentes. Para cada uno de los componentes se realiza una consulta a una serie de datos asociados a él. Estos datos se encuentran en una estructura aparte y cada uno de ellos es accedido a través de varios componentes. La computación realizada consiste en una reducción de histograma sobre los datos. El acceso a los datos se realiza a través de un índice en el caso de la versión vectorial y a través de un puntero en la versión dinámica. Se han empleado así mismo dos versiones del conjunto de datos, una (sin orden) obtenida a partir de un código generador y otra (con orden) tras aplicar un procedimiento de reordenación externo al conjunto

anterior. Observamos que las características de localidad del código basado en arrays son mejores que las de la versión basada en punteros.

Reordenar cómputos y datos sin modificar el resultado es también más difícil en este tipo de códigos. La reordenación de cómputos viene acotada por las limitaciones de los test de dependencias, sin cuyo conocimiento no es posible cambiar el orden de las sentencias con seguridad. Por otro lado la posibilidad de que varios punteros hagan referencia una misma posición de memoria, o que existan varios caminos en la estructura que lleven al mismo punto va a ser un problema cuando se persiga un reordenamiento de datos.

Este trabajo pretende analizar cómo el análisis de forma, una de las herramientas usadas en el contexto de códigos con punteros, permite facilitar el desarrollo de técnicas de compilación orientadas a la explotación de localidad. Básicamente el análisis de forma intenta representar la configuración en memoria de una estructura de datos basadas en punteros, definida sobre el *heap*, mediante un grafo reducido que plasme las caracterísitcas de la estructura. El grafo de forma se obtiene mediante una ejecución simbólica del código y es una representación no sólo de las posibles configuraciones en memoria de la estructura, sino que aporta valiosas propiedades asociadas a los nodos y aristas de dicho grafo. De las diversos análisis de forma que encontramos en la literatura emplearemos el desarrollado por Corbera [5], dada las ventajas que aporta sobre otros existentes ([6], [7], [8], [9]).

El resto del trabajo se ha organizado como sigue: En primer lugar se clasifican los códigos basados en punteros en función de los aspectos que se pueden tener en cuenta a la hora de explotar localidad. En segundo lugar se repasan algunas de las técnicas descritas en la literatura. Por último se presenta nuestro trabajo en curso, centrado en cómo las técnicas de análisis de forma pueden ayudar a mejorar la explotación de localidad en códigos basados en punteros.

II. Características de los códigos basados en punteros

Podríamos pensar que la explotación de localidad viene dada en gran medida por el orden en que el programa atraviesa los datos de las estructuras. Por tanto, en general, se dice que un código exhibe una buena localidad cuando los datos ocupan posiciones en memoria en un orden afín al orden en que serán accedidos. En códigos regulares que trabajan con vectores y matrices, este orden viene dado por el empaquetamiento usado en el almacenamiento de estas estructuras y su recorrido que sigue generalmente un patrón lineal. Sin embargo en códigos con estructuras con punteros la situación es diferente. En este apartado se discuten varias características relacionadas con el ordenamiento de los datos y que influirán en la estrategia a seguir.

	Basado en Arrays	Basado en Punteros
Con Orden	1.90% (L1) 0.32% (L2)	5.74% (L1) 1.25% (L2)
Sin Orden	3.90% (L1) 0.30% (L2)	8.59% (L1) 2.06% (L2)

TABLA I
Porcentaje de fallos de cache con respecto al número total de referencias para un código ejemplo que realiza una reducción de histograma.

En un código típico que maneje estructuras con punteros podemos distinguir dos fases en su ejecución. En primer lugar, la creación de la estructura de datos, y en segundo lugar, su uso. Durante la creación de la estructura se establece la ubicación inicial en memoria de los datos que la forman. Asímismo se determina el grafo sobre el que se definirán los caminos sobre la estructura, precisamente haciendo uso de los selectores o campos puntero.

Uno de los factores que pueden dirigir nuestra estrategia es la forma en que la estructura de datos cambia durante la ejecución. El caso más simple corresponde a una estructura estática, que una vez creada, no se modifica. En esta situación podríamos plantearnos una estrategia basada en el paradigma inspector–ejecutor. El inspector extraería las características de la estructura durante la ejecución, y podría realizar sobre ella transformaciones orientadas a la localidad, como una reubicación (*layout*). Esta acción se realizaría una sola vez, tras la creación de las estructuras y antes de su uso. La situación se complica si durante la ejecución se efectúan cambios en la estructura, es decir, si la estructura fuera dinámica. Ahora la ejecución puede modificar la estructura e incluso añadir nuevos elementos o borrar los existentes. Es precisamente este motivo el que puede llevar a un programador a usar una estructura de datos basada en punteros. Una estructura dinámica desaconserjaría la estrategia inspector–ejecutor, ya que implicaría una gran sobrecarga computacional adicional. No obstante podría seguirse esta aproximación si el inspector se aplicara periódicamente cuando se superara cierto umbral de cambio.

Otro aspecto a considerar es cómo la estructura se recorre durante su uso. En general los elementos de la estructura han sido ubicados en posiciones concretas durante la creación. Observamos dos hechos con respecto al uso de la estructura. El primero es que el orden en que se crean puede no coincidir con el recorrido, ni siquiera su ubicación seguir dicho recorrido. En segundo lugar una estructura puede ser recorrida de varias maneras: la estructura de punteros representan caminos posibles a seguir, pero serán unos en particular los que se sigan. Un ejemplo de esto lo constituye el caso de los múltiples recorridos en los que habitualmente se puede recorrer una estructura tipo árbol. La información de recorrido es

importante en la extracción de localidad, y por tanto debemos de recabar información no sólo de cómo es la estructura sino como vamos a navegar dentro de ella.

Un factor también a identificar es la operación predominante en el uso de la estructura, es decir, si la computación predomina sobre el recorrido de la estructura o viceversa. En problemas en los que la mayor parte del tiempo se consume en el recorrido será difícil realizar ocultamiento de la latencia de memoria, puesto que la estructura representa enlaces sucesivos a través de los punteros y acceder a elementos alejados supone una cadena sucesiva de referencias indirectas que no podremos solapar con el cálculo, al ser éste pequeño. Así mismo, en los problemas donde predomine la computación, el esfuerzo computacional asociado a una reubicación de los datos en memoria será menos significativo y podremos, además, realizarlos más frecuentemente, en caso de problemas dinámicos.

III. Estado-del-arte

Aunque, de forma general, no encontramos un método totalmente automático para optimizar el aprovechamiento de la localidad en programas que manejan estructuras basadas en punteros, sí podemos hallar en la literatura diversas aproximaciones, más o menos manuales, que persiguen este objetivo. En esta sección realizamos un repaso a dichas técnicas.

Chilimbi y otros [10] han analizado en qué puntos, desde que se escribe un código basado en punteros hasta su ejecución, se ha de considerar la localidad, proponiendo diversos métodos orientadas a su mejora. Las técnicas están orientadas a computadores con memoria cache, es decir, son técnicas conscientes de la existencia de la memoria cache (*cache-conscious*). La idea básica es tratar de organizar las estructuras manera que se maximicen los aciertos en las caches. Esto se puede enfocar desde cuatro niveles diferentes: en la definición de las estructuras de datos, en la reserva de memoria donde se ubicarán los datos, mediante transformaciones (reubicación) de la propia estructura de datos y durante las operaciones de liberación de memoria (*garbage collection*).

Centrándonos en los tres primeros niveles, que afectan directamente al programador, Chilimbi utiliza tres aproximaciones: empaquetado (*clustering*), coloreado, y compresión. La primera técnica agrupa los elementos de la estructura de manera que aquéllos que vayan a ser más probablemente accedidos estén en la misma línea cache. Por ejemplo, en un árbol se podría empaquetar el padre y sus hijos en la misma línea. No obstante, el resultado puede verse afectado por el recorrido concreto y la modificación, a posteriori, de la propia estructura.

El coloreado persigue evitar los conflictos debidos al nivel de asociatividad de la cache. Los elementos de la estructura más accedidos se colocan en líneas que no den lugar a conflictos al ser trasladadas a la cache, y los menos accedidos en el resto de líneas. Se trata de evitar que los conflictos desplacen datos frecuentemente accedidos.

En la compresión la estructura de datos se modifica, de forma que trabajamos sobre otra diferente a la original. Esto se realiza de manera que el mayor número de elementos de la estructura se compriman en una misma línea cache. Ahora, la forma de acceder a la nueva estructura cambia, es decir, para acceder a los datos habrá que descomprimir. Un ejemplo de esto podría ser el compactar varios elementos consecutivos de una lista enlazada en una misma línea cache. Ahora, a estos elementos se accederían con un desplazamiento sobre el primero, en lugar de seguir la cadena de enlaces de la estructura original.

Reservar memoria, para ubicar los datos, es una etapa que define inicialmente la posición de los elementos y por tanto afecta directamente a la localidad. En general, el programador realiza esta reserva mediante llamadas a una función tipo *malloc*, despreocupándose de dónde estarán físicamente los elementos. En [10] se plantea diseñar funciones de reserva de memoria que sean conscientes de la jerarquía de memoria (*cache-concious allocation* o *cc-malloc*). Así, reservaremos posiciones que estarán cerca de las que probablemente vamos a acceder. Esto implica trasladar al compilador un conocimiento de la semántica del código, y de la forma en que los punteros recorrerán la estructura.

Adicionalmente, puesto que la estructura de datos se puede recorrer en un orden diferente al que se crea y, además, puede cambiar durante la ejecución, las técnicas de ubicación conscientes de la cache se pueden reforzar con una reubicación dinámica que también sea consciente de la cache. En [10] se propone un reestructurador (*cc-morph*) que permite reorganizar óptimamente una estructura tipo árbol, aplicando empaquetado y coloreado. En este caso, la información semántica que requiere el compilador es menor, ya que sólo necesita saber el punto de entrada a la estructura, parámetros simples topológicos y los parámetros de configuración de la cache.

Una técnica ampliamente usada en códigos regulares basados en *arrays* con el objetivo de ocultar la latencia es el *prefetching*. En [11] se explora la posibilidad de emplear esta técnica en códigos con punteros. El objetivo es resolver el problema conocido como *pointer chasing*. Este problema se plantea para una estructura de datos recursiva, en la que se accede a un elemento después de recorrer una cadena de punteros, y consiste en determinar el valor de ese elemento de forma que se oculte la latencia de memoria, es decir, que el tiempo en determinar cuál es el siguiente elemento sea menor que el tiempo de computación. Una solución consiste en usar los denominados *jump-pointers* que actúan como atajos, que permiten acceder a un elemento determinado sin recorrer la cadena completa de punteros que determina el camino. Por ejemplo, en una lista enlazada existe un puntero al siguiente elemento. Un *jump-pointer* consistiría en añadir

enlaces a elementos posteriores al siguiente.

Estos punteros pueden existir ya de por sí en la estructura (naturales) o son introducidos *ad hod* artificialmente para ocultar la latencia. En [11], Luk y otros proponen diversas técnicas de *prefetching* basadas en *jump-pointers*, como el *history-pointer prefetching* o el *greedy prefeching*. En el primero se crean enlaces artificiales que determinan el recorrido real sobre la estructura, para lo cual se emplea una cola histórica de posiciones recorridas. El segundo consiste en realizar un *prefetching* de los nodos que son directamente apuntados por un elemento de la estructura.

Como extensión del trabajo [11], Karlsson y otros generalizan la técnica de *jump-pointers* empleando vectores de *prefetching* [12]. Esta idea consiste en asociar un vector de *jump-pointers* que apuntan a los nodos que serán visitados en las subsiguientes sentencias a ejecutar. Dicho vector se guarda junto con cada nodo, para maximizar la localidad. Además, se propone un hardware específico que permite un *prefetching* por bloques, que trata de anticipar, de una sola vez, los datos apuntados por los elementos de un vector de *prefeching*.

En general, el *prefetching* está limitado por la incapacidad de predecir con antelación cuál es el siguiente elemento de la estructura que se va a utilizar. No obstante, en trabajos como [13], [14], se intenta descubrir regularidades existentes en las referencias irregulares de las aplicaciones que usan punteros. Esta regularidad puede ser descubierta empleando técnicas estadísticas mediante un *profiling* en tiempo de ejecución. La información estadística se usa para modificar el programa añadiendo las sentencias de *prefetching* que tienen en cuenta las referencias a realizar en instantes futuros.

En [13] se presenta una técnica capaz de identificar *strides* en el patrón de acceso. Para ello se identifican las instrucciones de carga (*load*) indirectas correspondientes a accesos a estructuras a través de punteros. Tras ejecutar el código instrumentado, se recopilan datos de las primeras y segundas diferencias de las direcciones accedidas por estas cargas, a partir de los cuales se predice el *stride* a usar en los *prefetching*.

En [14] se describe una representación eficiente de las referencias de datos que permite explotar localidad. Esta representación es varios órdenes de magnitud más pequeña que la traza completa del programa. Se basa en un algoritmo jerárquico de compresión y permite descubrir la regularidad de los patrones irregulares, entendiendo por regularidad la repetición de subsecuencias en el patrón de acceso. La representación del patrón obtenida durante la ejecución permite descubrir secuencias de accesos que poseen regularidad (*hot data streams*). A partir de esta localización se aplican técnicas de *clustering* y *prefetching*.

Junto con las técnicas software implementables en un compilador encontramos propuestas orientadas a hardware específico como en [15], [16]. El diseño propuesto localiza dependencias entre cargas que producen una dirección y cargas que las consumen. Estos pares de cargas representan una acceso a través de una indirección, como ocurre al atravesar una estructura de datos basada en punteros. Los pares son identificados en una ventana de ejecución y se almacenan en una tabla, denominada de correlación, que mantendrá aquellos pares productor/consumidor usados más recientemente. La información de esta tabla se emplea para realizar especulativamente un *prefetching* cada vez que se ejecute una instrucción de carga productora.

IV. Explotación de localidad y análisis de forma

En esta sección se estudia cómo el análisis de forma, una de las herramientas de compilación empleadas a la hora de describir las estructuras basadas en punteros, puede ayudar a la automatización de las técnicas orientadas a localidad. En particular, el análisis de forma al que vamos a hacer referencia es el desarrollado por Corbera en [5], dado que aporta ventajas significativas sobre otros análisis propuestos en la literatura ([7], [8], [9]).

Este análisis, tras una ejecución simbólica del código, genera un conjunto de grafos, para cada sentencia del programa, que sintetizan la configuración de memoria asociada a la estructura de datos, tal como existe en ese punto de ejecución. Sale de los objetivos de este artículo describir en detalle dicho análisis, del que destacamos dos ideas básicas. Por un lado, la estructura de datos queda reflejada en un conjunto reducidos de grafos que denominaremos RSRSG (*Reduced Set of Reference Shape Graph*), donde cada nodo representa un conjunto de elementos con propiedades de acceso equivalentes. Por otro lado, el proceso de generación del RSRSG nos proporciona información adicional asociada a los nodos y conexiones que forman el grafo.

Esta información adicional de los nodos expresa características deducibles asociadas a las posiciones de memoria que representan. Podemos citar, ya que haremos uso de ella, la propiedad *shared*, que indica si un nodo puede ser accedido desde diferentes punteros o selectores de otros nodos. De forma ilustrativa, en la figura 1 se muestra los RSRSG asociados a diferentes puntos de la ejecución de un código que inserta elementos en una lista simplemente enlazada.

En particular nos vamos a centrar en la ubicación de los datos de cara a un mejor uso de la jerarquía de memoria. Esto lo vamos a hacer desde las dos vertientes comentadas en la sección previa: la reserva de espacio para los datos y la reubicación de las estructuras, una vez creadas, que sean conscientes de la jerarquía de memoria.

Con respecto a la primera aproximación, vamos a considerar que disponemos de una función tipo *malloc* consciente de la jerarquía de memoria (la denotaremos `hc-malloc`). Esta función es capaz de proporcionar, bajo petición, una zona de memoria siguiendo determinadas políticas de localidad. Esto

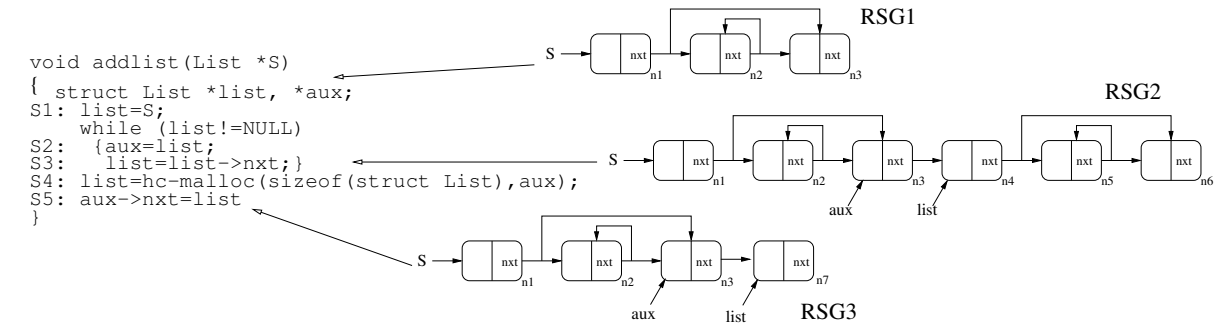

Fig. 1. Añadiendo un elemento al final de una lista simplemente enlazada.

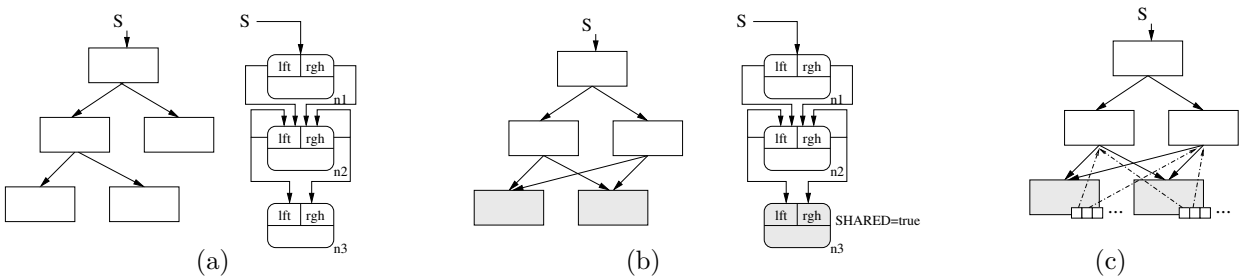

Fig. 2. *Back jump-pointers* asociados a nodos compartidos (*shared*).

supone que conoce los parámetros de la jerarquía y que al realizar la reserva de espacio va a proporcionar una porción de memoria cercana. Este concepto de cercanía va a depender de la estructura, y es, en este punto, donde el análisis de forma es útil. Por ejemplo, en [10] se plantea una función de reserva que utiliza un puntero como parámetro de la función. La porción de memoria reservada ha de ser cercana a dicho puntero. Así, por ejemplo, una llamada a `hc-malloc(size, *p)` indicará reservar un bloque cercano al puntero `p`. El principal inconveniente de esta técnica es la necesidad de conocer, por parte del programador, la semántica del código, ya que hay que elegir de qué puntero queremos estar cerca, lo cual es difícil sin conocer la aplicación. No obstante automatizar la conversión de `malloc` convencionales a `hc-malloc` puede realizarse con la ayuda análisis de forma. Vamos a ver esto en un ejemplo que consiste en añadir un elemento al final de una lista simplemente enlazada.

En la figura 1 se muestra una función que añade un elemento al final de una lista simple, así como un grafo de forma ilustrativo de las posibles configuraciones de forma en diferentes sentencias. A través de un puntero auxiliar se localiza el punto de inserción, se reserva memoria para un nuevo nodo y se añade a la lista. En este caso, se pretende que el nuevo nodo esté cerca del último, para lo cual la función que reserva memoria, con conocimiento de la jerarquía, va a requerir como parámetro dicha posición. Se puede observar cómo el grafo de forma nos aporta esta información de cercanía entre punteros.

Cuando lo que consideramos es una reorganización de la estructura, la información proporcionada por el análisis de forma también es útil. Consideremos un reorganizador consciente de la jerarquía de memoria, similar a la función `cc-morph` propuesta en [10]. Varios de los parámetros topológicos necesarios por este reestructurador pueden ser obtenidos de la información de forma. En primer lugar, podemos saber si la función de transformación es aplicable, al reconocer si se trata de una estructura específica como un árbol, una lista, ... En segundo lugar, se puede extraer información topológica de forma automática. Así, por ejemplo, fijándonos en una estructura tipo árbol, a las que son aplicables el transformador `cc-morph`, el análisis de forma nos aportaría varios datos importantes, como el punto de entrada, el número de hijos de los nodos, y la función de acceso de los hijos desde el padre.

Una limitación importante de la reorganización consciente de la jerarquía de memoria está en los elementos apuntados por varios selectores. Durante la transformación, cuando uno de estos nodos cambia su ubicación, habría que modificar todos los punteros que apuntan a él, para que la reestructuración no modifique el grafo que representa la estructura. De aquí surge, precisamente, la limitación de la función `cc-morph`, que es sólo es aplicable cuando el grafo tiene un único punto de entrada y un nodo no es accedido por varios selectores (estructuras de tipo arbóreo).

Entre las propiedades proporcionadas para los nodos del grafo de forma generado según [5] podemos destacar la capacidad de conocer si un nodo de la estructura es *shared*, en el sentido de si puede ser apuntado por diferentes punteros pertenecientes a otros nodos. Al analizar la forma de una estructura, esta propiedad nos da información de si un nodo está referenciado por varios punteros ó selectores. De este modo, podríamos generalizar la función de reorganización de árboles a estructuras con nodos *shared*, o con varios puntos de entrada usando esta información. Para ello, necesitamos llevar constancia de quién referencia un nodo *shared*, ya que tenemos localizadas las sentencias en las que se establece un puntero a dicho nodo. Esto se puede hacer asociando al nodo *shared*, una lista de punteros hacia detrás

(*back jump-pointer*) que apunten a los nodos que son padres.

En la figura 2 se muestra un ejemplo. Tomando un árbol binario como el mostrado en (a) su forma asociada no contendrá ningún nodo shared. No obstante, si los nodos hojas finales pudieran tener varios padres, entonces daría lugar a nodos *shared*, como refleja el análisis de forma mostrado en (b). El caso (a) permite una reubicación simple ya que un nodo es apuntado por un sólo padre. Sin embargo, en el caso (b), la posibilidad de reubicar las hojas es más difícil, ya que hay nodos que pueden apuntarse por varios padres y, al recorrer la estructura y llegar por un camino, sólo tenemos conocimiento de uno de ellos. La forma nos sugiere donde añadir los *back jump-pointers* que facilitan la reorganización de la estructura y en que sentencias debemos asignarles valores.

V. Conclusiones

El aprovechamiento de la localidad es un aspecto clave a la hora de obtener un buen rendimiento de códigos que trabajan con estructuras de datos basadas en punteros. Este problema no está completamente resuelto de una manera general, dado el desconocimiento, tanto de la forma de la estructura como de las posiciones concretas en las que se emplazará en memoria en tiempo de ejecución.

En este artículo se han analizado los aspectos a tener en cuenta en este tipo de códigos y se ha hecho un repaso de diferentes estrategias que han sido propuestas. Por último, hemos demostrado que el análisis de forma puede ser de gran ayuda al diseñar técnicas automáticas orientadas a explotar localidad.

Referencias

[1] S. D. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz, "Run-time and compile-time support for adaptative irregular problems," in *Proceedings Supercomputing'94*, November 1994, pp. 97–106.

[2] Hao Yu and Lawrence Rauchwerger, "Adaptive reduction parallelization techniques," in *ICS 2000*, Santa Fe, New Mexico, USA, May 2000, pp. 66–77.

[3] Hwansoo Han and Chau-Wen Tseng, "Efficient compiler and run-time support for parallel irregular reductions," *Parallel Computer*, 2000.

[4] Eladio Gutierrez, Oscar Plata, and Emilio L. Zapata, "A compiler method for the parallelization of iregular reductions on scalable shared memory multiprocessors," in *Proceedings of the 2000 ACM International Conference on Supercomputing*, May 2000.

[5] F. Corbera, R. Asenjo, and E.L. Zapata, "A framework to capture dynamic data structures in pointer-based codes," *IEEE Trans. on Parallel and Distributed System*, vol. 15, no. 2, pp. 151–166, 2004.

[6] J. Plevyak, A. Chien, and V. Karamcheti, "Analysis of dynamic structures for efficient parallel execution," in *Languages and Compilers for Parallel Computing*, Berlin, 1993, pp. 37–57, Sprienger-Verlag.

[7] R. Ghiya and L.J. Hendren, "Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C," in *23rd ACM Symposium on Principles of Programming Languages*, 1996, pp. 1–15.

[8] M. Sagiv, T.W. Reps, and R. Wilhelm, "Parametric shape analysis via 3-valued logic," in *Symposium on Principles of Programming Languages*, 1999, pp. 105–118.

[9] T. Lev-Ami and M. Sagiv, "TVLA: A system for implementing static analyses," in *Static Analysis Symposium*, 2000, pp. 280–301.

[10] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus, "Making pointer-based data structures cache conscious," *Computer*, vol. 33, no. 12, December 2000.

[11] Chi-Keung Luk and Todd C. Mowry, "Compiler-based prefetching for recursive data structures," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, October 1996.

[12] Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom, "A prefetching technique for irregular accesses to linked data structures," in *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, 2000, pp. 206–217.

[13] Youfeng Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, June 2002.

[14] Trishul M. Chilimbi, "Efficient representations and abstractions for quantifying and exploiting data reference locality," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, June 2001.

[15] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, October 1998.

[16] Amir Roth and Gurindar S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *Proceedings of the 26th annual international symposium on Computer architecture*, May 1999.