Accurate Shape Analysis for Recursive Data Structures

F. Corbera
R. Asenjo
E.L. Zapata

August 2000
Technical Report No: UMA-DAC-00/33

Published in:

13th Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'2000)
IBM T.J. Watson Res. Ctr., Yorktown Heights, New York, NY
August 10-12, 2000

University of Malaga
Department of Computer Architecture
C. Tecnologico • PO Box 4114 • E-29080 Malaga • Spain

Accurate shape analysis for recursive data structures*

F. Corbera, R. Asenjo, and E. Zapata

Dept. Computer Architecture, University of Málaga, Spain
{corbera,asenjo,ezapata}@ac.uma.es

Abstract. Automatic parallelization of codes which use dynamic data structures is still a challenge. One of the first steps in such parallelization is the automatic detection of the dynamic data structure used in the code. In this paper we describe the framework and the compiler we have implemented to capture complex data structures generated, traversed, and modified in C codes. Our method assigns a *Reduced Set of Reference Shape Graphs* (RSRSG) to each sentence to approximate the shape of the data structure after the execution of such a sentence. With the properties and operations that define the behavior of our RSRSG, the method can accurately detect complex recursive data structures such as a doubly linked list of pointers to trees where the leaves point to additional lists. Other experiments are carried out with real codes to validate the capabilities of our compiler.

1 Introduction

For complex and time-consuming applications, parallel programming is a must. Automatic parallelizing compilers are designed with the aim of dramatically reducing the time needed to develop a parallel program by generating a parallel version from a sequential code without special annotations. There are several well-known research groups involved in the development and improvement of parallel compilers, such as Polaris, PFA, Parafrase, SUIF, etc. We have noted that the detection step of current parallelizing compilers does a pretty good job when dealing with regular or numeric codes. However, they cannot manage irregular codes or symbolic ones, which are mainly based on complex data structures which use pointers in many cases. Actually, data dependence analysis is quite well known for array-based codes even when complex array access functions are present [6]. On the other hand, much less work has been done to successfully determine the data dependencies of code sections using dynamic data structures based on pointers. This is mainly because of the huge complexity implicit in this problem. Nevertheless, this is a problem that cannot be avoided due to the increasing use of dynamic structures and memory pointer references.

With this motivation, our goal is to propose and implement new techniques that can be included in compilers to allow the automatic parallelization of real codes based on dynamic data structures. From this goal we have selected the shape analysis subproblem, which aims at estimating at compile time the shape the data will take at run time. Given this information, a subsequent analysis would detect whether or not certain sections of the code can be parallelized because they access independent data regions.

There are several ways this problem can be approached. The simplest strategy consists in asking the programmer to annotate the sequential code, so helping the compiler with this extra information [8] following a semiautomatic approach. On the other hand, user interaction is avoided in many studies, such as the ones based on "access paths" or on graphs. For example, the method proposed by Matsumoto et al. [10] uses "normalized" path expressions to maintain the "alias-pair" between pointers. However, we discarded this set

* This work was supported by the Ministry of Education and Science (CICYT) of Spain (TIC96-1125-C03), by the European Union (BRITE-EURAM III BE95-1564), by APART: Automatic Performance Analysis: Resources and Tools, EU Esprit IV Working Group No. 29488

of methods as they cannot handle cyclic structures such as double linked lists or trees with pointers from leaves to parents.

On the other hand, in the graph-based methods the "storage chunks" are represented by nodes, and edges are used to represent references between them. For example, Chase et al. [3] define the "Storage Shape Graph" which contains one node for each variable and one for each allocation site in the program. With this abstraction their method can detect a single linked list even when new items are appended to the end of the list. However, this method is not powerful enough to detect insertion of elements in the middle of the list. Plevyak et al. [11] try to solve Chase's problem by extending the "Storage Shape Graphs" to the "Abstract Storage Graph" (ASG). However, the improvements in accuracy are paid for with too much complexity in comparisons and compression operations. On the other hand, the method presented by Sagiv et al. [12] is based on what they call "Static Shape Graphs" (SSG). The main difference between this method and the previous ones lie in the node-name scheme they use for the nodes, where all the memory locations not directly pointed to by a pointer variable are fused into a single summary node.

In a previous work [4], we combined and extended Plevyak and Sagiv's methods allowing more than a summary node per graph among other extensions. However, we keep the restriction of one graph per sentence in the code. This way, since each sentence of the code can be reached after following several paths in the control flow, the associated graph should approximate all the possible memory configurations arising after the execution of this sentence. This restriction leads to memory and time saving, but at the same time it significantly reduces the accuracy of the method. In this work, we have changed our previous direction by selecting a tradeoff solution: we consider several graphs with more than a summary node, while fulfilling some rules to avoid an explosion in the number of graphs and nodes in each graph.

Among the first relevant studies which allowed several graphs were those developed by Jones et al. [9] and Horwitz et al. [7]. These approaches are based on a "k-limited" approximation in which all nodes beyond a k selectors path are joined in a summary node. The main drawback to these methods is that the node analysis beyond the "k-limit" is very inexact and therefore they are unable to capture complex data structures. A more recent work that also allows several graphs and summary nodes is the one presented by Sagiv et al.[13]. They propose a parametric framework based on a 3-valued logic. To describe the memory configuration they use 3-valued structures defined by several predicates. These predicates determine the accuracy of the method. As far as we know the currently proposed predicates do not suffice to deal with the complex data structures that we handle in this paper.

With this in mind, our proposal is based on approximating all the possible memory configurations that can arise after the execution of a sentence by a set of graphs: the *Reduced Set of Reference Shape Graphs* (RSRSG). We see that each RSRSG is a collection of *Reference Shape Graphs* (RSG) each one containing several non-compatible nodes. Finally, each node represents one or several memory locations. Compatible nodes are "summarized" into a single one. Two nodes are compatible if they share the same reference properties. With this framework we can achieve accurate results without excessive compilation time. Besides this, we cover situations that were previously unsolved, such as detection of complex structures (lists of trees, lists of lists, etc.) and structure permutation, as we will see in this article.

The rest of the paper is organized as follows. Section 2 briefly describes the whole framework, introducing the key ideas of the method and presenting the data structure example that will help in understanding node properties and operations with graphs. These properties are described in Sect. 3 where we show how the RSG can accurately approximate a memory configuration and when two RSGs are compatible. The analysis method have been implemented in a compiler which is experimentally validated, in Sect. 4, by analyzing several

C codes based on complex data structures. Finally, we summarize the main contributions and future work in Sect. 5.

2 Method overview

Basically, our method is based on approximating all possible memory configurations that can appear after the execution of a sentence in the code. Note that due to the control flow of the program, a sentence could be reached by following several paths in the control flow. Each "control path" has an associated memory configuration which is modified by each sentence in the path. Therefore, a single sentence in the code modifies all the memory configurations associated with all the control paths reaching this sentence. Each memory configuration is approximated by a graph we call *Reference Shape Graphs* (RSG). So, taking all this into account, we conclude that each sentence in the code will have a set of RSGs associated with it. This set of RSGs will describe the shape of the data structure after the execution of this sentence.

The calculation of this set of graphs is carried out by the **symbolic execution** of the program over the graphs. In this way, each program sentence transforms the graphs to reflect the changes in the memory configurations derived from the sentence execution. The RSGs are graphs in which nodes represent memory locations which have similar reference patterns. Therefore, a single node can safely and accurately represents several memory locations (if they are similarly referenced) without losing their essential characteristics.

To determine whether or not two memory locations should be represented by a single node, each one is annotated with a set of properties. Now, two different memory locations will be "summarized" in a single node if they fulfill the same properties. Note that the node inherits the properties of the memory locations represented by this node. Besides this, two nodes can be also summarized if they represent "summarizable" memory locations. This way, a possibly unlimited memory configuration can be represented by a limited size RSG, because the number of different nodes is limited by the number of properties of each node. These properties are related to the reference pattern used to access the memory locations represented by the node. Hence the name *Reference Shape Graph*.

As we have said, all possible memory configurations which may arise after the execution of a sentence are approximated by a set of RSGs. We call this set *Reduced Set of Reference Shape Graphs* (RSRSG), since not all the different RSGs arising in each sentence will be kept. On the contrary, several RSGs related to different memory configurations will be fused when they represent memory locations with similar reference patterns. There are also several properties related to the RSGs, and two RSGs should share these properties to be joined. Therefore, besides the number of nodes in an RSG, the number of different RSGs associated with a sentence are limited too. This union of RSGs greatly reduces the number of RSGs and leads to a practicable analysis.

The symbolic execution of the code consists in the abstract interpretation of each sentence in the code. This abstract interpretation is carried out iteratively for each sentence until we reach a fixed point in which the resulting RSRSG associated with the sentence does not change any more [5]. This way, for each sentence that modifies dynamic structures, we have to define the abstract semantics which describes how these sentences modify the RSRSG. We consider six simple instructions that deal with pointers: $x = NULL$, $x = malloc$, $x = y$, $x \rightarrow sel = NULL$, $x \rightarrow sel = y$, and $x = y \rightarrow sel$. More complex pointer instructions can be built upon these simple ones and temporal variables.

The output RSRSG resulting from the abstract interpretation of a sentence over an input RSRSG_i is generated by applying the abstract interpretation to each $rsg_i \in \text{RSRSG}_i$. Due to space constraints we cannot formally describe the abstract semantics of each one of these sentences. However, we intuitively present the modifications involved in an RSG after the sentence symbolic execution:

- The $x = NULL$ sentence leads to elimination of the references from the pointer variable (pvar) x to any memory location. Therefore we have to remove these references from the RSG.
- The $x = malloc$ sentence simply initializes new memory locations represented in the RSG by a node referenced by x.
- The $x = y$ modifies the RSG such that all memory locations pointed to by y are now also pointed to by x. Before this sentence and the previous one, we always automatically insert the $x = NULL$ sentence to ensure that before assigning a new value to x, x is not pointing to any place.
- The sentence $x \rightarrow sel = NULL$ is the most complex one, because it break links between nodes. This leads to many changes in the properties of the nodes. In order to obtain an accurate output RSG before removing the $x \rightarrow sel$ link we divide the RSG into several rsg_j. This division is carried out by taking into account that each rsg_j should have a single destination (node) for the $x \rightarrow sel$ link. In this way we can better focus on the several memory configurations represented by the RSG regarding this $x \rightarrow sel$ link. Each rsg_j is pruned after the division to remove redundant or inexistent nodes or links which have been conservatively inherited from the parent RSG. We also increase the accuracy of the method, materializing from the node the memory location which is the real target of the $x \rightarrow sel$ link. After this, this link can be safely removed.
- The $x \rightarrow sel = y$ sentence implies the execution of the same procedure just described for the $x \rightarrow sel = NULL$ sentence (RSG division, pruning, and node materialization) followed by the generation of a link by selector sel from the nodes pointed to by x to the nodes pointed to by y.
- Finally, the sentence $x = y \rightarrow sel$ leads to the inclusion of a new reference from the x pvar to all the memory locations pointed to by $y \rightarrow sel$. Now, the RSG division, pruning, and node materialization are carried out for the $y \rightarrow sel$ link. In this way we will point with x to the exact memory locations pointed to by $y \rightarrow sel$ and to no other.

After the abstract interpretation of the sentence over the $rsg_i \in \text{RSRSG}_i$ we obtain a set of output rsg_o. As we said, we cannot keep all the rsg_o arising from the abstract interpretation. On the contrary, each rsg_o will be compressed, which means the summarization of compatible nodes in the rsg_o. Furthermore, some of the rsg_os can be fused in a single RSG if they represent similar memory configurations. This operation greatly reduces the number of RSGs in the resulting RSRSG. The similarity between memory configurations is defined, among other properties, by the alias relations between pointer variables.

More specifically, our analysis method has been designed to always avoid the fusion of two different RSGs, rsg_1 and rsg_2, if a pointer variable x points to different non-compatible nodes in each RSG (in rsg_1 $x \rightarrow n_1$ and in rsg_2 $x \rightarrow n_2$). In other words, in our framework, a pvar can only point to a single node in a certain RSG. This situation concords with the fact that for a certain sentence in the code, a pvar cannot point to two different memory locations. However, due to the control flow of the code, we allow an RSG for each control path due to the fact that for a sentence, a pvar can point to different places when following different control paths.

All the operations just enumerated in this section (RSG division, pruning, node summarization, joining of compatible RSG, etc.) are described in the next two sections. In order to better illustrate these operations we present a data structure example which will be refered to during the framework and operations description.

2.1 Working example

The data structure, presented in Fig. 1, is a doubly linked list of pointers to trees. Besides this, the leaves of the trees have pointers to doubly linked lists. The pointer variable S points

to the first element of the doubly linked list (header list). Each item in this list has three pointers: *nxt*, *prv*, and *tree*. This *tree* selector points to the root of a binary tree in which each element has the lft and *rgh* selectors. Finally, the leaves of the trees point to additional doubly linked lists. All the trees pointed to by the header list are independent and do not share any element. In the same way, the lists pointed to by the leaves of the same tree or different trees are also independent.

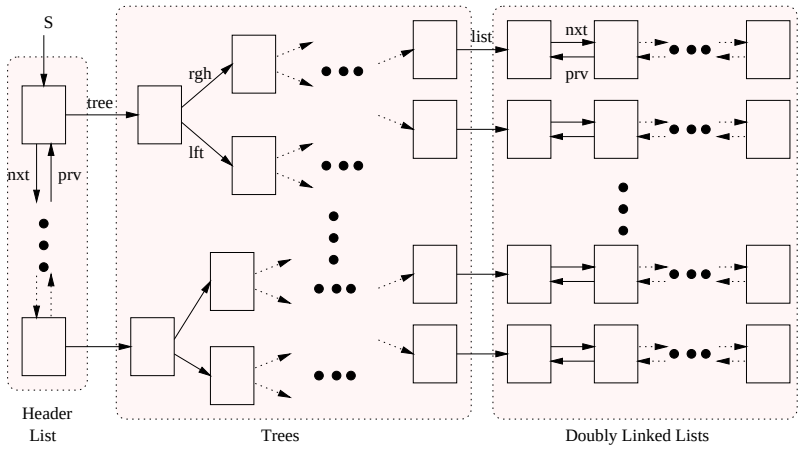


Fig. 1. A complex data structure.

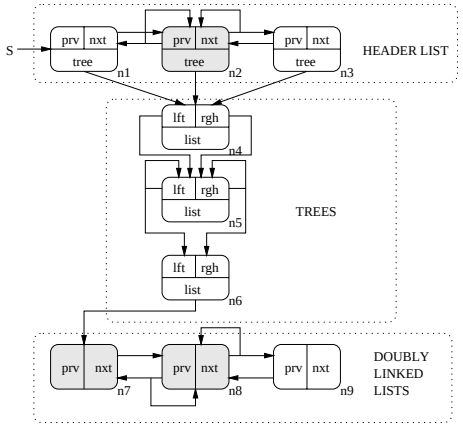
This data structure is built by a C code using the data structure declaration shown in figure 2 (a). This code also traverses the elements of the header list with two pointers and eventually can permute two trees. Our compiler has analyzed this code obtaining an RSRSG for each sentence in the program. Figure 2 (b) shows a compact represetation of the RSRSG obtained for the last sentence of the code after the compiler analysis.

```
typedef struct doubly_linked_list {
  /* non-pointer fields definition */
  struct doubly_linked_list *nxt, *prv;
} doubly_linked_list;

typedef struct tree {
  /* non-pointer fields definition */
  struct tree *lft, *rgh;
  struct doubly_linked_list *list;
} tree;

typedef struct header_list {
  /* non-pointer fields definition */
  struct header_list *nxt, *prv;
  struct tree *tree;
} header_list;
```

(a)



(b)

Fig. 2. Structure declaration and compact representation of the resulting RSRSG for the data structure of Fig. 1.

As we will see in the next sections, from the RSRSG represented in Fig. 2 (b) we can infer the actual properties of the real data structure: the trees and lists do not share elements and therefore they can be traversed in parallel. These results and other examples of real codes (sparse matrix-vector multiplication, sparse LU factorization and Barnes-Hut N-body simulation) with different data structures are presented in Sect. 4. But first, we describe our framework with a formal description of the RSGs in the next section, followed by the definition and operations of the RSRSG in Sect. 3.8.

3 Reference Shape Graph

First, we need to present the notation used to describe the different memory configurations that may arise when executing a program.

Definition 1 *We call a collection of dynamic structures a memory configuration. These structures comprise several memory chunks, that we call memory locations, which are linked by references. Inside these memory locations there is room for data and for pointers to other memory locations. These pointers are called selectors.*

We represent the memory configurations with the tuple $M = (L, P, S, PS, LS)$ *where:* **L** *is the set of memory locations;* **P** *is the set of pointer variables (pvars) used in the program;* **S** *is the set of selectors declared in the data structures;* **PS** *is the set of references from pvars to memory locations, of the type* $< pvar, l >$*, with* $pvar \in P$ *and* $l \in L$*; and* **LS** *is the set of links between memory locations, of the form* $< l_1, sel, l_2 >$ *where* $l_1 \in L$ *references* $l_2 \in L$ *by selector* $sel \in S$*.*

We will use L(m), P(m), S(m), PS(m), and LS(m) to refer to each one of these sets for a particular memory configuration, m. □

Therefore, we can assume that the RSRSG of a program sentence is an approximation of the memory configuration, M, after the execution of this sentence. But let us first describe the *RSGs* now that we know how a memory configuration is defined.

Definition 2 *An RSG is a graph represented by the tuple* $RSG = (N, P, S, PL, NL)$ *where:* **N***: is the set of nodes. Each node can represent several memory locations if they fulfill certain common properties;* **P***: is the set of pointer variables (pvars) used in the program;* **S***: is the set of declared selectors;* **PL***: is the set of references from pvars to nodes, of the type* $< pvar, n >$ *with* $pvar \in P$ *and* $n \in N$*; and* **NL***: is the set of links between nodes, of the type* $< n_1, sel, n_2 >$ *where* $n_1 \in N$ *references* $n_2 \in N$ *by selector* $sel \in S$*.*

We will use N(rsg), P(rsg), S(rsg), PL(rsg), and NL(rsg) to refer to each one of these sets for a particular RSG, rsg. □

Note that the P(m) and P(rsg) sets are the same in both memory and graph domains, so from now on we will refer to them as P. The same can be applied to S(m) and S(rsg).

To obtain the RSG which approximates the memory configuration, M, an abstraction function is used, $F : M \rightarrow RSG$. This function maps memory locations into nodes and references to memory locations into references to nodes at the same time. In other words, F, translates the memory domain into the graph domain. This, function F comprises three functions: $F_n : L \rightarrow N$ takes care of mapping memory locations into nodes; $F_p : PS \rightarrow PL$ maps references from pvars to memory locations into references from pvars to nodes, and $F_l : LS \rightarrow NL$ maps links between locations into links between nodes.

Proposition 1 *It is easy to see that:*

- $F_p(< pvar, l >) =< pvar, n >$ *iif* $F_n(l) = n$
- $F_l(< l_1, sel, l_2 >) =< n_1, sel, n_2 >$ *iif* $F_n(l_1) = n_1 \wedge F_n(l_2) = n_2$

which means that translating references to locations into references to nodes is trivial after mapping locations into nodes. This translates almost all the complexity involved in function F to function F_n which actually maps locations into nodes. □

Now we focus on F_n which extracts some important properties from a memory location and, depending on these, this location is translated into a node. Besides this, if several memory locations share the same properties then this function maps all of them into the same node of the *RSG*. Due to this dependence on the location properties, the F_n function will be described during the presentation of the different *properties* which characterize each node. These properties are: Type, Structure, Reference pattern, Share information, and Cycle links. These are now described.

3.1 Type

This property tries to extract information from the code text. The idea is that two pointers of different types should not point to the same memory location (for example, a pointer to a graph and another to a list). Also, the memory location pointed to by a graph pointer should be considered as a graph. This way, we assign the `TYPE` property, of a memory location l, from the type of the pointer variable used when that memory location l is created (by malloc or in the declarative section of the code).

Therefore, two memory locations, l_1 and l_2, can be mapped into the same node if they share the same `TYPE` value: If $F_n(l_1) = F_n(l_2) = n$ then $\texttt{TYPE}(l_1) = \texttt{TYPE}(l_2)$, where the `TYPE()` function returns the `TYPE` value. Note that we can use the same property name and function, `TYPE()`, for both memory locations and nodes. Clearly, $\texttt{TYPE}(n) = \texttt{TYPE}(l)$ when $F_n(l) = n$.

This property leads to the situation where, for the data structure presented in Fig. 1, the nodes representing list memory locations will not be summarized with those nodes representing tree locations, as we can see in Fig. 2 (b).

3.2 Structure

As we have just seen, the `TYPE` property keeps two different nodes for two memory locations of different types. However, we also want to avoid the summarization of two nodes which represent memory locations of the same type but which do not share any element. That is, they are non-connected components. This behavior is achieved by the use of the `STRUCTURE` property, which takes the same value for all memory locations (and nodes) belonging to the same connected component. Again, two memory locations can be represented by the same node if they have the same `STRUCTURE` value: If $F_n(l_a) = F_n(l_b) = n$ then $\texttt{STRUCTURE}(l_a) = \texttt{STRUCTURE}(l_b)$

This leads to the condition that two locations must fulfill in order to share the same `STRUCTURE` value: $\texttt{STRUCTURE}(l_a) = \texttt{STRUCTURE}(l_b) = val$ iif $\exists l_1, ..., l_i | (< l_a, sel_1, l_1 >, < l_1, sel_2, l_2 >, ..., < l_i, sel_{i+1}, l_b > \in LS) \vee (< l_b, sel_1, l_1 >, < l_1, sel_2, l_2 >, ..., < l_i, sel_{i+1}, l_a > \in LS)$, which means that two memory locations, l_a and l_b, have the same `STRUCTURE` value if there is a path from l_a to l_b (first part of the previous equation) or from l_b to l_a (second part of the equation). In the same way we can define $\texttt{STRUCTURE}(n)$.

3.3 Simple paths

The `SPATH` property further restricts the summarizing of nodes. *Simple paths* denominates the access path from a pointer variable (pvar) to a location or node. An example of a simple path is $p \rightarrow s$ in which the pvar p points to the location s. In this example the simple path for s is $< p >$.

The use of the simple path avoids the summarization of nodes which are directly pointed to by the pvars, and therefore these nodes are the entry points to the data structure. We define the SPATH property for a memory location $l \in L(m)$ as $\texttt{SPATH}(l) = p_1, .., p_n$ where $< p_i, l > \in PS(m)$. This property is similarly defined for the RSG domain. Now, two memory locations are represented by the same node if they have the same SPATH (If $F_n(l_1) = F_n(l_2)$ then $\texttt{SPATH}(l_1) = \texttt{SPATH}(l_2)$).

3.4 Reference patterns

This new property is introduced to classify and represent by different nodes the memory locations with different reference patterns. We understand by reference pattern the type of selectors which point to a certain memory location and which point from this memory location to others.

This is particularly useful for keeping singular memory locations of the data structure in separate nodes. For example, the head/tail and the rest of the elements of a single linked list are two kinds of memory locations. These will be represented by different nodes, because the head location is not referenced by other list entries and the tail location does not reference any other list location. The same would also happen for more complex data structures built upon more simple structures (such as lists of lists, trees of lists, etc.). For example, in Fig. 1, the root of one of the trees is referenced by the header list and the leaves do not point to tree items but to a doubly linked list. Thanks to the reference patterns, the method results in the RSRSG of Fig. 2 (b), where the root of the tree, the leaves, and the other tree items are clearly identified.

In order to obtain this behavior, we define two sets SELINset and SELOUTset which contain the set of input/output selectors for a certain location: $\texttt{SELINset}(l_1) = \{sel_i \in S | \exists l_2 \in L, < l_2, sel_i, l_1 > \in LS\}$ and $\texttt{SELOUTset}(l_1) = \{sel_i \in S | \exists l_2 \in L, < l_1, sel_i, l_2 > \in LS\}$, where we see that sel_i is in the $\texttt{SELINset}(l_1)$ if l_1 is referenced from somewhere by selector sel_i, or sel_i is in $\texttt{SELOUTset}(l_1)$ if $l_1.sel_i$ points to somewhere outside.

3.5 Share information

This is a key property for informing the compiler about the potential parallelism exhibited by the analyzed data structure. Actually, the share information can tell whether at least one of the locations represented by a node is referenced more than once from other memory locations. That is, a shared node represents memory locations which can be accessed from several places and this may prevent the parallelization of the code section which traverses these memory locations. From another point of view, this property helps us to determine if a cycle in the RSG graph is representing cycles in the data structure approximated by this RSG or not.

Due to the relevance of this property, we use two kinds of attributes for each node:

- $\texttt{SHARED}(n)$ with $n \in N(rsg)$, is a Boolean function which returns "true" if any of the locations, l_1, represented by n are referenced by other locations, l_2 and l_3, by different selectors, sel_i and sel_j. Therefore, this SHARED function tells us whether there may be a cycle in the data structure represented by the RSG or not. If $\texttt{SHARED}(n)$ is 0, we know that even if we reach the node n by sel_1 and later by sel_2, we are actually reaching two different memory locations represented by the same n node, and therefore there is no cycle in the approximated data structure.
- $\texttt{SHSEL}(n, sel)$ with $n \in N(rsg)$ and $sel \in S$, is a Boolean function which returns "true" if any of the memory locations, l_1, represented by n can be referenced more than once by selector sel from other locations, l_2 and l_3. This way, with the SHSEL function, we can distinguish two different situations that can be represented by an RSG with a node,

n, and a selector, sel, pointing to itself. If $\texttt{SHSEL}(n, sel) = 0$ we know that this node is representing an acyclic unbounded data structure (the size is not known at compile time). For example, in a list, all the elements of the list (locations) are represented by the same node, n, but following selector sel we always reach a different memory location. On the other hand, if $\texttt{SHSEL}(n, sel) = 1$, for the same list example, by following selector sel we can reach an already visited location, which means that there are cycles in the data structure.

Let's illustrate these SHARED and SHSEL properties using the compact representation of the RSRSG presented in Fig. 2 (b). In this Fig., shaded nodes have the SHARED property set to true. For example, in the header list the middle node n_2 is shared, $\texttt{SHARED}(n_2)$=1, because n_2 is referenced by selectors nxt and prv. However, the $\texttt{SHSEL}(n_2, nxt)$=$\texttt{SHSEL}(n_2, prv)$ =0 which means that by following selector nxt or prv it is not possible to reach an already visited memory location. Actually, in this example, there are no selectors with the SHSEL property set to true. So, the same happens for node n_8 which represents the middle items of the doubly linked lists.

We can also see in Fig. 2 (b), that node n_4 is not shared, which states that, in fact, from memory locations represented by n_1, n_2, and n_3 we can reach different trees which do not share any elements (as we see in Fig. 1). Finally, node n_7 is shared because it is pointed to by selectors $list$ and prv. However, due to $\texttt{SHSEL}(n_7, list)$=0 we can ensure that two different leaves of the trees will never point to the same doubly linked list.

3.6 Cycle links

The goal of this property is to increase the accuracy of the data structure representation by avoiding unnecessary edges that can appear during the RSG updating process.

The cycle links of a node, n, are defined as the set of pairs of references $< sel_i, sel_j >$ such that when starting at node n and consecutively following selectors sel_i and sel_j, the n node is reached again. More precisely, for $n \in N(rsg)$ we define: $\texttt{CYCLELINKS}(n) = \{< sel_i, sel_j > | sel_i, sel_j \in S\}$, such that if $< sel_i, sel_j > \in \texttt{CYCLELINKS}(n)$ then: $\forall l_i, F_n(l_i) = n$, if $< l_i, sel_i, l_j > \in LS$ then $\exists < l_j, sel_j, l_i > \in LS$

This CYCLELINKS set maintains similar information to that of "identity paths" in the Abstract Storage Graph (ASG) [11], which is very useful for dealing with doubly-linked structures. For example, in the data structure presented in Fig. 1, the elements in the middle of the doubly linked lists have two cycle links: $< nxt, prv >$ and $< prv, nxt >$, due to starting at a list item and consecutively following selectors nxt and prv (or prv and nxt) the starting item is reached. Note, that this does not apply to the first or last element of the doubly linked list. This property is captured in the RSRSG shown in Fig. 2 (b) where we see three nodes for the double linked lists (one for the first element of the list, another for the last element, and another between them to represent the middle items in the list). This middle node, n_8, is annotated by our compiler with $\texttt{CYCLELINKS}(n_8) = \{< nxt, prv >, < prv, nxt >\}$.

We conclude here that the CYCLELINKS property is used during the pruning process which take place after the node materialization and RSG modification. So, in contrast with the other four properties described in previous subsections, the CYCLELINKS property does not prevent the summarization of two nodes with do not share the CYCLELINKS sets and therefore do not affect the F_n function.

3.7 Compression of graphs

After the symbolic execution of a sentence over an input RSRSG, the resulting RSRSG may contain RSGs with redundant information, which can be removed due to node summarization or compression of the RSG.

In order to do this, after the symbolic execution of a sentence, the method applies the COMPRESS function over the just modified RSGs. This COMPRESS function first call to the boolean C_NODES_RSG one, which identifies the compatible nodes that will later be summarized. This Boolean function just has to check whether or not the first five properties previously described are the same for both nodes (as we said in the previous subsection the CYCLELINKS property does not affect the compatibility of two nodes).

There is a similar function which returns true when two memory locations are compatible. With this, we can finally define F_n as the function which maps all the compatible memory locations into the same node, which happens when they have the same TYPE, STRUCTURE, SHARED and SPATH properties, and compatible reference patterns.

3.8 Reduced Set of RSGs

We have already seen that an RSG describes a memory configuration by a finite graph. The execution of each sentence in the program can modify the memory configuration and thus the RSG. However, due to the control flow of the program, the same sentence could be reached by several control paths leading to several memory configurations depending on the path. This way, there could also be several RSGs for the same program sentence. In our method, we maintain the representation of all these RSGs with the *Reduced Set of Reference Shape Graphs* (RSRSG).

Therefore, each sentence in the program has an associated RSRSG which approximately describes all possible memory configurations that may arise after the execution of this sentence.

However, the number of RSGs stored in an RSRSG could be prohibitive if we do not apply some simplifications while keeping reasonable accuracy in the representation. Actually, this simplification consists in allowing the union of some of the RSGs which fulfill certain conditions. After the union, the resulting RSG should represent all the locations approximated by the original RSGs and the relevant shape information should be maintained.

More precisely, two graphs, rsg_1 and rsg_2, can be joined in a single one if they are compatible which hapens when they fulfill two conditions: i) the alias relation between pvars of both graphs are the same; and ii) certain nodes in both graphs are compatible. This leads us to define the alias relation between pvars and the compatibility condition between certain nodes in the graphs:

- ALIAS(rsg) is the set of alias relations, alr_i, in the rsg graph, where each alr_i identifies all the pvars pointing to the same node n_i.
- COMP_NODES(rsg_1, rsg_2) is a Boolean function which returns true if the nodes directly pointed to by the same pvar are compatible, which happens when they have similar properties (and therefore these nodes can be summarized).

Let us note again that we only impose node compatibility on those nodes directly pointed to by the same pvar. On the other hand, this node compatibility is fulfilled when two nodes have the same TYPE, SHARED and SHSEL properties and they have a similar reference pattern.

Finally, there are several operations that can be carried out with the RSGs of an RSRSG. In the worst case, the sequence of operations that the compiler carries out in order to symbolically execute a sentence are: graph division, graph prune, sentence symbolic execution (RSG modification), RSG compression and RSG union to build the final RSRSG. In fact, the RSG compression has already been summarized in Sect. 3.7. However, the graph division, prune and union operations cannot be described in this paper due to space constraints.

4 Experimental results

All the previously mentioned operations and properties have been implemented in a compiler written in C which analyzes a C code to generate the RSRSG associated with each sentence

of the code. As we said, the symbolic execution of each sentence over an RSRSG is going to generate a modified RSRSG. Before the symbolic execution of the code, the compiler can also extract some important information from the program in a previous pass. For example, a quite frequent pattern arising in C codes based on dynamic data structures is the following: `while (x != NULL) { ... }`.

In this case the compiler can assert that at the entry of the while body the pvar $x \neq NULL$. Besides this, if we have not exited the while body with a `break` sentence, we can also ensure that just after the while body the pvar $x = NULL$. This information is used to simplify the analysis and increase the accuracy of the method. More precisely, we can reduce the number of RSGs and/or reduce the complexity of this RSG by diminishing the number of memory configurations represented by each RSG. Other sentences from which we can extract useful information are `IF-THEN-ELSE`, `FOR` loops, or any conditional sentence.

The implementation of this idea has been carried out by the definition of certain *pseudoinstructions* that we call `FORCE`. These pseudoinstructions are inserted in the code by the first pass of the compiler and will be symbolically executed as a regular sentence. Therefore, each one of these `FORCE` sentences has its own abstract semantics and its own associated RSRSG. The `FORCE` pseudoinstructions we have considered are: $\texttt{FORCE}_{[x==NULL]}(rsg)$, $\texttt{FORCE}_{[x!=NULL]}(rsg)$, $\texttt{FORCE}_{[x==y]}(rsg)$, $\texttt{FORCE}_{[x!=y]}(rsg)$, $\texttt{FORCE}_{[x\rightarrow sel==NULL]}(rsg)$.

In addition, we have found that these `FORCE` pseudoperations can be also placed just before the following sentences: $x \rightarrow sel = NULL$, $x \rightarrow sel = y$ y $y = x \rightarrow sel$, under the assumption that the code is correct. That is, it makes sense to assume that before the execution of all these three sentences, x is not NULL (in other cases the code would produce an error at execution time). The same can be assumed for any sentence with an occurrence of the type $x \rightarrow val$, where val is a non-pointer field of the structure pointed to by x. In these cases, we can safely insert just before these sentences the pseudoinstruction $\texttt{FORCE}_{[x!=NULL]}$.

With the compiler described we have analyzed the code which generates, traverses, and modifies several codes: the working example presented in section 2.1, the Sparse Matrix-vector multiplication, the Sparse LU factorization and the Barnes-Hut code. All these codes were analyzed by our compiler in a Pentium III 500MHz with 128MBytes main memory. The time and memory required by the compiler are summarized in table 1. The particular aspects of these codes are described next.

	Working Example	S. Matrix-vector	S. LU factorization	Barnes-Hut
Time	0'13"	0'04"	1'38"	4'04"
Space	2.7 MB	1.6 MB	24 MB	47 MB

Table 1. Time and space required by the compiler to analyze several codes

4.1 Working example's RSRSG

We refer in this subsection to the code that generates, traverses, and modifies the data structure presented in Fig. 1. A compact representation of the resulting RSRSG for the last sentence of the code can be seen in Fig. 2 (b). Although we do not show the code due to space constraints, we have to say that this code presents an additional difficulty due to some tree permutations being carried out during data structure modification. The problem arising during structure permutation is that it is very easy to temporally assign the `SHARED`=true property to the root of one of the trees that we are permutating, when this root is temporally pointed to by two different locations from the header list. If this shared property remains true after the permutation we would have a shaded n_4 node in Fig. 2 (b). This would imply that

two different items from the header list can point to the same tree (which would prevent the parallel execution of traversing the trees). However, this problem is solved because, after the permutation, the method reassigns false to the shared property thanks to the combination of our properties and the division of graph operations. Summarizing, after the compiler analyzes this code, the compact representation of the resulting RSRSG for the last sentence of the program (Fig. 2 (b)) accurately describes the data structure depicted in Fig. 1 in the sense that: (i) The compiler successfully detects the doubly linked list which is acyclic by selectors nxt or prv and whose elements point to binary trees; (ii) As $\texttt{SHSEL}(n_4, tree)$=0, we can say that two different items of the header list cannot point to the same tree; (iii) At the same time, as no tree node (n_4, n_5 and n_6) is shared, we can say that different trees do not share items; (iv) The same happens for the doubly linked list pointed to by the tree leaves: all the lists are independent, there are no two leaves pointing to the same list, and these lists are acyclic by selectors nxt or prv.

Besides this, our compiler has also analyzed three C program kernels which generate, traverse, and modify complex dynamic data structures which we describe next.

4.2 Sparse matrix-vector multiplication

Here we deal with an irregular code which implements a sparse matrix by vector multiplication, $r = M \times v$. The sparse matrix, M, is stored in memory as a header doubly linked list with pointers to other doubly linked lists which store the matrix rows. The sparse vectors, v and r are also doubly linked lists. This can be seen in Fig. 3(a). Note that vector r grows during the multiplication process.

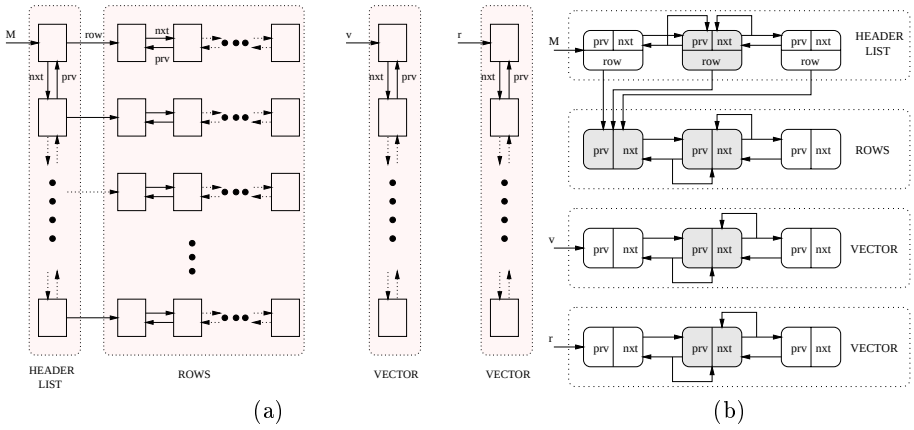


Fig. 3. Sparse matrix-vector multiplication data structure and compacted RSRSG.

After the analysis process, carried out by our compiler, the resulting RSRSG accurately represents this data structure. Actually, in Fig. 3(b) we present a compact representation of the resulting RSRSG for the last sentence of the code. First, note that the three structures involved in this code are kept in separate subgraphs. Even when the data type for vectors v and r and rows of M, is the same, the `STRUCTURE` property avoids the union of these graphs into a single one. This RSRSG states that the rows of the matrix are pointed to from different elements of the header list (there is no selector with the shared property set to true). Also, the doubly linked lists which store the rows of M and the vectors v and r are acyclic by selectors nxt and prv.

The same RSRSG is also reached just before the execution of the outermost loop which takes care of the matrix multiplication, but without the r subgraph which is generated during this multiplication.

4.3 Sparse LU factorization

The kernel of many computer-assisted scientific applications is to solve large sparse linear systems. We find examples of these kinds of applications in optimization problems, linear programming, simulation, circuit analysis, fluid dynamics computation, and numeric solutions of differential equations in general. The code we analyze now is one of these solvers: a sparse LU factorization. This algorithm presents a good case study and is a representative computational code for many other irregular problems. Actually, this problem represents those in which the computational load grows with the execution time (fill-in) and matrix coefficients change their coordinates due to row/column permutations (pivoting).

More specifically, our working example application solves non-symmetric sparse linear systems by applying the LU factorization of the sparse matrix, computed by using a general method [1]. These methods directly solve the sparse problem and share the same loop structure of the corresponding dense code. In particular, we have implemented an in-place code for the direct right-looking LU algorithm, where an n-by-n matrix A is factorized. The code includes a row pivoting operation (partial pivoting) to provide numerical stability and preserve sparseness. The input matrix A columns as well as the resulting in place LU columns are stored in one-dimensional doubly linked lists (see Fig. 4 (a)), to facilitate the insertion of new entries and to allow column permutations.

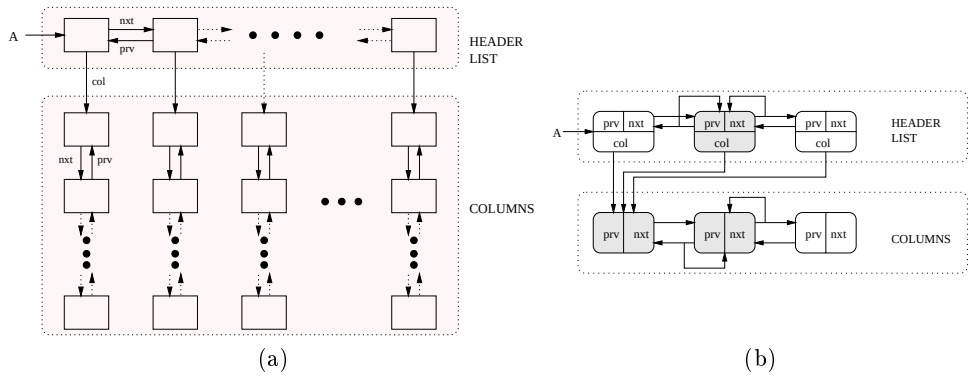


Fig. 4. Sparse LU factorization data structure and compacted RSRSG.

After the LU code analysis we obtain the same RSRSG for the sentences just after the matrix initialization and after the LU factorization. A compact representation of this RSRSG is shown in Fig. 4(b). As we can see, variable A points to a doubly linked list, the header list. Each node of this list points to a single doubly linked list which represents a matrix column. Due to `SHSEL` being false for all selectors we conclude that the header list and the column lists are acyclic structures when they are traversed by a single selector type and that different elements of the header list should point to different columns. A subsequent analysis of the code and the RSRSG associated with each sentence would be able to state that several sparse matrix columns can be updated in parallel during factorization and, in addition, it is also possible to update each column in parallel.

4.4 Barnes-Hut N-body simulation

This code is based on the algorithm presented in [2] which is used in astrophysics. In fact, this application simulates the evolution of a system of bodies under the influence of gravitational forces. It is a classical gravitational N-body simulation, in which each body is modeled as a point mass. The simulation proceeds over time-steps, each step computing the net force on every body and thereby updating that body's position and other attributes. The data structure used in this code is based on a hierarchical octree representation of space in three dimensions. In two dimensions, a quadtree representation is used. However, due to memory constraints (the octree and the quadtree versions run out of memory in our 128MB Pentium III) we have simplified the code to use a binary tree.

In Fig. 5(a) we present a schematic view of the data structure used in this code. The bodies are stored by a single linked list pointed to by the pvar *Lbodies*.

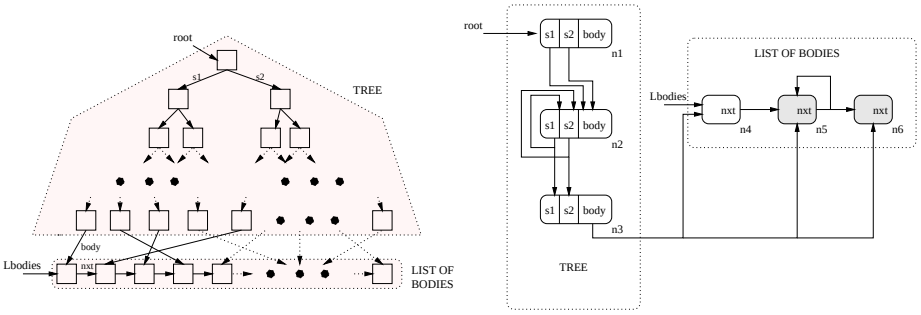


Fig. 5. Barnes-Hut data structure and compacted RSRSG.

After the code analysis, the compact representation of the RSRSG at the end of each step of the algorithm is presented in Fig. 5(b). We can see that the root of the tree is represented by node n_1, the middle elements of the tree by node n_2 and the leaves by n_3. Note that these leaves can point to any body stored in the *Lbodies* list represented by nodes n_4, n_5, and n_6. As tree nodes are not shared and selectors also have the `SHSEL` property set to false, a subsequent analysis of the code can state that the tree can be traversed and updated in parallel. This analysis can also conclude that there are no two different leaves pointing to the same body (entry in the *Lbodies* list) due to nodes n_4, n_5, and n_6 not being shared by selector *body*.

5 Conclusions and future work

We have developed a compiler which can analyze a C code to determine the RSRSG associated with each sentence of the code. Each RSRSG contains several RSGs, each one representing the different data structures which may arise after following different paths in the control flow graph of the code. However, several RSGs can be joined if they represent similar data structures, in this way reducing the number of RSGs associated with a sentence. Every RSG contains nodes which represent one or several memory locations. To avoid an explosion in the number of nodes, all the memory locations which are similarly referenced are represented by the same node. This reference similarity is captured by the properties we assign to the memory locations. In comparison with previous works, we have increased the number of properties assigned to each node. This leads to more nodes in the RSG because

the nodes now have to fulfill more properties to be summarized. However, by avoiding the summarization of these nodes, we keep a more accurate representation of the data structure. This is a key issue when analyzing the parallelism exhibited by a code.

Our compiler symbolically executes each sentence in the code, transforming the RSGs to reflect the modifications in the data structure that are carried out due to the execution of the sentence. We have validated the compiler with several C codes which generate, traverse, and modify complex dynamic data structures, such as a doubly linked list of pointers to trees where the leaves point to other doubly linked lists. Other structures have been also accurately identified by the compiler, even in the presence of structure permutations (for example, column permutations in the sparse LU code). As far as we know, there is no compiler achieving such successful results for these kinds of data structures appearing in real codes.

In the near future we will develop an additional compiler pass that will automatically analyze the RSRSGs and the code to determine the parallel loops of the program and allow the automatic generation of parallel code.

References

1. R. Asenjo. Sparse LU Factorization in Multiprocessors. Ph.D. Dissertation, Dept. Computer Architecture, Univ. of Málaga, Spain, 1997.
2. J. Barnes and P. Hut. *A Hierarchical O(n· log n) force calculation algorithm.* Nature v.324, December 1986.
3. D. Chase, M. Wegman and F. Zadeck. *Analysis of Pointers and Structures.* In SIGPLAN Conference on Programming Language Design and Implementation, 296-310. ACM Press, New York, 1990.
4. F. Corbera, R. Asenjo and E.L. Zapata *New shape analysis for automatic parallelization of C codes.* In ACM International Conference on Supercomputing, 220–227, Rhodes, Greece, June 1999.
5. P. Cousot and R. Cousot. *Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points.* In Proceedings of the ACM Symposium on Principles of Programming Languages. ACM Press, New York. 238–252, 1977.
6. J. Hoeflinger and Y. Paek *The Access Region Test.* In Twelfth International Workshop on Languages and Compilers for Parallel Computing (LCPC'99), The University of California, San Diego, La Jolla, CA USA, August, 1999.
7. S. Horwitz, P. Pfeiffer, and T. Reps. *Dependence Analysis for Pointer Variables.* In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, 28-40, June 1989.
8. J. Hummel, L. J. Hendren and A. Nicolau *A General Data Dependence Test for Dynamic, Pointer-Based Data Structures.* In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pages 218-229. ACM Press, 1994.
9. N. Jones and S. Muchnick. *Flow Analysis and Optimization of Lisp-like Structures.* In Program Flow Analysis: Theory and Applications, S. Muchnick and N. Jones, Englewood Cliffs, NJ: Prentice Hall, Chapter 4, 102-131, 1981.
10. A. Matsumoto, D. S. Han and T. Tsuda. *Alias Analysis of Pointers in Pascal and Fortran 90: Dependence Analysis between Pointer References.* Acta Informatica 33, 99-130. Berlin Heidelberg New York: Springer-Verlag, 1996.
11. J. Plevyak, A. Chien and V. Karamcheti. *Analysis of Dynamic Structures for Efficient Parallel Execution.* In Languages and Compilers for Parallel Computing, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua, Eds. Lectures Notes in Computer Science, vol 768, 37-57. Berlin Heidelberg New York: Springer-Verlag 1993.
12. M. Sagiv, T. Reps and R. Wilhelm. *Solving Shape-Analysis problems in Languages with destructive updating.* ACM Transactions on Programming Languages and Systems, 20(1):1-50, January 1998.
13. M. Sagiv, T. Reps, and R. Wilhelm, *Parametric shape analysis via 3-valued logic.* In Conference Record of the Twenty-Sixth ACM Symposium on Principles of Programming Languages, San Antonio, TX, Jan. 20-22, ACM, New York, NY, 1999, pp. 105-118.