Progressive Shape Analysis for Real C Codes

F. Corbera
R. Asenjo
E.L. Zapata

September 2001
Technical Report No: UMA-DAC-01/09

Published in:

IEEE Int'l. Conf. on Parallel Processing (ICPP'2001)
Valencia, Spain, September 3-7, 2001

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

Progressive Shape Analysis for Real C Codes *

F. Corbera
Computer Architecture Dept.
University of Málaga. Spain.
corbera@ac.uma.es

R. Asenjo
Computer Architecture Dept.
University of Málaga. Spain.
asenjo@ac.uma.es

E. Zapata
Computer Architecture Dept.
University of Málaga. Spain.
ezapata@ac.uma.es

Abstract

Dynamic and pointer-based data structures are widely used in symbolic or irregular C codes. However, there is still a lack of compiler techniques to deal with the automatic optimization of such codes. In this paper we take a first step towards this final objective: the automatic identification of the data structure used in the code. More precisely, 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. The compiler makes a progressive analysis in which the level of detail is increased during the analysis when needed. Several experiments are carried out with complex data structures to validate the capabilities of our compiler.

1. Motivation

Although languages like C, C++, Java, etc., have been widely used for many years, there is still a lack of compiler techniques able to automatically optimize the execution of real codes written in these languages. The difficulties which have prevented the development of such techniques lie mainly in the huge complexity associated with pointers and dynamic data structures. In this sense, for Fortran77 codes – in which these data structures are not allowed – there are a large number of optimizing and parallelizing compilers which successfully deal with complex code transformation even in the presence of irregular array access [3]. However, new solutions are a must due to the wide acceptance of these programming languages and due to the fact that dynamic data structures are important tools to achieve good performance and simplify the development of complex codes.

With this motivation, our goal is to propose and implement new techniques that can be included in compilers to allow the automatic analysis of real codes based on dynamic data structures. In a first step, 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, but we focus on the graph-based methods in which the "storage chunks" are represented by nodes, and edges are used to represent references between them [6], [7], [1]. These methods use only one graph to approximate the memory after the execution of each sentence. In this work, we consider several graphs for each sentence.

Among the first relevant studies which allowed several graphs were those developed by Jones et al. [5] and Horwitz et al. [4], however they are unable to capture complex data structures. A more recent work that also allows several graphs is the one presented by Sagiv et al.[8]. As far as we know, their method can not 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.

The rest of the paper is organized as follows: Sect. 2 briefly describes the whole framework, introducing the key ideas of the method. The node properties and operations with graphs are described in Sect. 3. In Sect. 4 we present the operations related with the RSGs included in an RSRSG. These operations have been implemented in a

*This work was supported by the Ministry of Education and Science (CICYT) of Spain (TIC 2000-1658), by the European Union (BRITE-EURAM III BE95-1564), by APART: Automatic Performance Analysis: Resources and Tools, EU Esprit IV Working Group No. 29488

compiler which is experimentally validated, in Sect. 5. Finally, we summarize the main contributions and future work in Sect. 6.

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. Each memory configuration is approximated by a graph we call *Reference Shape Graphs* (RSG). Thus, 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 represent 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. 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.

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 the *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. As we will see, 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. This way, for each sentence that modifies dynamic structures, we have to define the abstract semantics which describe 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. Due to space constraints we cannot describe the abstract semantics of each one of these sentences (see [2]).

However, we can present a simple example. In Fig. 1 (a) we see an RSG representing a doubly linked list of two or more elements. Actually, n_1 represents the first element in the list, n_2 the middle elements, and n_3 the last one. Let us suppose that this RSGs is an input rsg_i to the $x \rightarrow nxt = NULL$ sentence. The first step in the abstract interpretation of this sentence is the *division* operation. Figure 1 (b) shows the resulting $rsg\prime_1$ and $rsg\prime_2$ after the division. Note that in each one of these graphs there is a single destination for $x \rightarrow nxt$. This division process is formally described in Sect. 4.1. In Fig. 1 (c) we show the result of the *pruning* process in which the compiler removes nodes and links which do not fulfill the graphs' properties. This pruning process is formally described in Sect. 4.2. Now, before removing the $x \rightarrow nxt$ link in both graphs, the compiler has to focus more on one of the RSGs. More precisely, in $rsg\prime\prime_1$, we have to *materialize* from node n_2 the node n_4 which represents the single list item referenced by $x \rightarrow nxt$, as we can see in Fig. 1 (d). Finally, we see in Fig. 1 (e) how we safely remove the link $x \rightarrow nxt$ in both graphs to obtain the final rsg_1 and rsg_2.

From a higher perspective, the whole symbolic execution process can be seen by looking at Fig. 2. For each sentence in the code, the RSRSG$_i$ comprises several rsg_{ij} to capture all the memory configurations associated with each path in the control flow graph. During the symbolic execution of the sentence all these rsg_{ij} are going to be updated. The first step comprises the graph *division* and *pruning* processes after which we obtain several rsg_{ijk}. Then the abstract interpretation of the sentence takes place and usually the complexity of the RSGs grows. In order to counter this effect, the compiler carries out a *compression* of the graph phase in which each RSG is simplified by the summarization of compatible nodes in the RSG, to obtain the rsg^*_{ijk} graphs. This step is formally described in Sect. 3.1. Furthermore, some of the rsg^*_{ijk}s can be fused into a single rsg_{ok} if they represent similar memory configurations. This operation greatly reduces the number of RSGs in the resulting RSRSG. This graph *union* is described in Sect. 4.3.

All the operations just enumerated in this section are described in the next two sections.

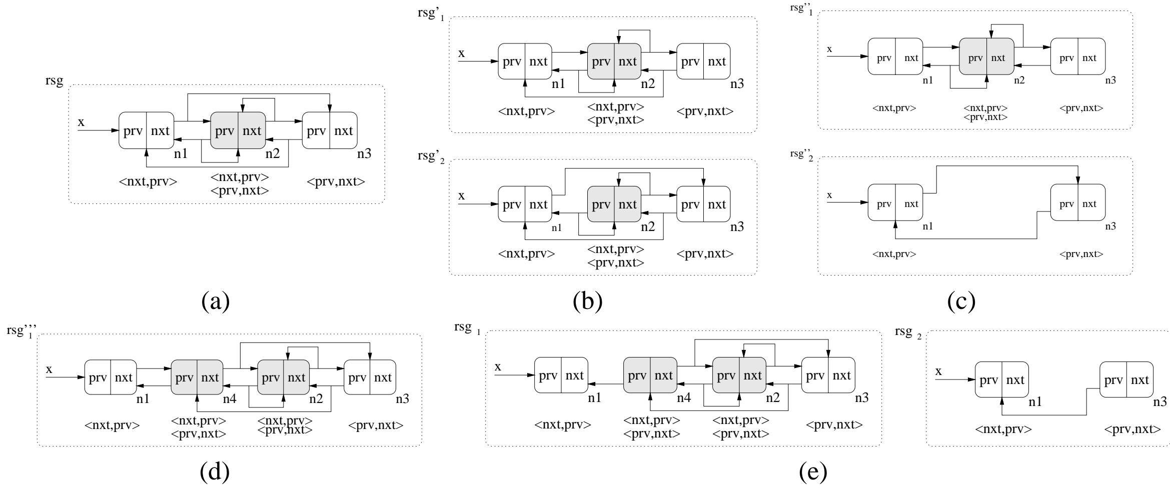

Figure 1. Complete process of the abstract interpretation required by the $x \to nxt = NULL$ sentence.

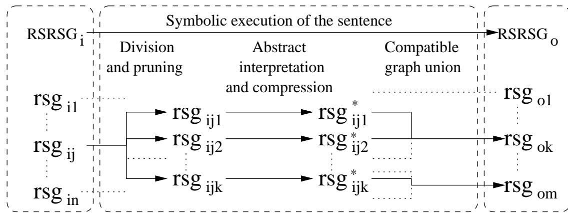

Figure 2. Schematic description of the symbolic execution of a sentence.

3. Reference Shape Graph

An RSG is a graph represented by the tuple $RSG = (N, P, S, PL, NL)$ where: **N**: is the set of nodes. **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 $sel \in S$.

To obtain the RSG which approximates a memory configuration, we extract some important properties from the memory locations and, depending on these, the locations are translated into nodes. Besides this, if several memory locations share the same properties then all of them are mapped into the same node of the RSG. These properties are: **Type** states the data type of the memory locations represented by a node; **Structure** avoids the summarization of nodes representing non-connected components; **Reference pattern** keeps singular memory locations of the data structure in separate nodes; **Share information** tell whether at least one of the locations represented by a node is referenced more than once from other memory locations; **Cycle links** keeps information about simple cycles in the structure. In [2] we present a complete description of these properties. We have modified the property **Simple paths** and we have inserted a new property **Touch information**. These two properties are now described:

Simple paths denominates the access path from a pointer variable (pvar) to a node if the length of this path is less than or equal to 1. An example of a simple path is $p \to s.sel \to t$ in which the pvar p points to node s which points to node t using the selector sel. Note that, in this example, the simple path for t is $< p, sel >$ and the simple path for s is $< p, \emptyset >$. The length of a simple path, $sp_i =< pvar, sel_i >$, $\texttt{LEN}(sp_i) = 0$ if $sel_i = \emptyset$ or 1 if $sel_i = sel \in S$.

To determine when two nodes can be summarized, we define a Boolean function $\texttt{C_SPATH}(n_1, n_2, m)$ which returns true if nodes n_1 and n_2 have compatible SPATH. The parameter m imposes the constraints in the comparison of nodes. If $m = 0$, two node SPATHs are compatible if they comprise the same zero-length simple paths. This particular case of the C_SPATH function is called C_SPATH0. On the contrary, if $m = 1$, the two SPATHs also have to share at least 1 one-length simple paths to be compatible. In this case the function is called C_SPATH1.

Touch information. All previous properties capture some important issues of the memory configuration and they change according to the current sentence. However, sometimes it is necessary to keep track of the memory locations for several sentences to increase the accuracy of the RSRSGs. For example, if we deal with a list data structure, we know that all the middle elements are going to be summarized in a single node. Now, when traversing this list in

a loop, the same summary node will represent non-visited locations as well as visited ones. On the other hand, during one acyclic traversal of the list, it would be better to keep the visited locations in a separate node in such a way that new changes only affect non-visited nodes.

In order to achieve this behavior in the compiler we assign to each node a new property called TOUCH. This property is taken into account only inside loop bodies. In this case, the TOUCH information of a node is the set of pvars from which the memory locations represented by the node have been visited. We understand by "pvar x visit node n" that the node n has been referenced by x. For example, in $x = y$, the node pointed to by y is visited by x. In the same way, in $x = y \rightarrow sel$, the node pointed to by selector sel from node y is visited by x.

Now, two nodes can be summarized if they have been "touched" by the same set of pvars. However, clearly this new restriction in the summarization will increase the number of nodes in the RSRSGs. In order to avoid the explosion in the number of nodes we have to constrain the kind of pvar which can appear in the TOUCH set. More precisely, only those pvars which are used to traverse dynamic data structures (called induction pointers by Yuan-Shin Hwang [9]) are eligible to be included in the set. Taking all this into account, we can finally define for $n \in N(rsg)$: $\mathtt{TOUCH}(n) = \{ipvar | ipvar \in iP\}$ where iP is the set of induction pvars found in the code. Clearly, there should be a preprocessing compiler pass to identify inductions pvars in the code. Due to space constraints we cannot describe this preprocessing pass but it is based on Access Path Expressions [9].

Finally, the compiler also implements an additional improvement to save space and time. The idea is that after exiting a loop body the TOUCH information regarding the ipvars of this loop are not needed any more. This way, the compiler removes those ipvars associated with this loop from the TOUCH set of the nodes.

3.1. Compression of graphs

As we explained in Sect. 2, 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. However, before explaining this COMPRESS function, we need to define the C_NODES_RSG one, which identifies the compatible nodes that will later be summarized. This Boolean function just has to check whether or not some of the properties are the same for both nodes. This way

$\mathtt{C_NODES_RSG}(n_i, n_j) = true$ if $(\mathtt{TYPE}(n_i) = \mathtt{TYPE}(n_j)) \wedge (\mathtt{STRUCTURE}(n_i) = \mathtt{STRUCTURE}(n_j)) \wedge (\mathtt{SHARED}(n_i) = \mathtt{SHARED}(n_j)) \wedge (\mathtt{SHSEL}(n_i, sel_k) = \mathtt{SHSEL}(n_j, sel_k) \forall sel_k \in S) \wedge (\mathtt{TOUCH}(n_i) = \mathtt{TOUCH}(n_j)) \wedge (\mathtt{C_REFPAT}(n_i, n_j) = 1) \wedge (\mathtt{C_SPATH}(n_i, n_j, m) = 1)$.

where SHARED and SHSEL keep the *shared information* of a node, and $\mathtt{C_REFPAT}(n_i, n_j)$ tell if both nodes have compatible *reference pattern* information (see [2]).

Now, compatible nodes of the same RSG are summarized by the function $\mathtt{COMPRESS}(rsg) = rsg_c$, where:

• The set of nodes of the compressed RSG, $N(rsg_c)$, will contain the nodes which cannot be summarized with any other plus the nodes resulting from the summarization of compatible nodes. We use the MERGE_COMP_NODES function to generate a summary node from a group of compatible nodes, as we will see later. Formally:

$$N(rsg_c) = \{n \mid (n \in N(rsg) \wedge (\nexists n_i \in N(rsg)) \wedge \mathtt{C_NODES_RSG}(n, n_i) = 1)) \vee (n = \mathtt{MERGE_COMP_NODES}(n_1, ..., n_k), n_1, ..., n_k \in N(rsg) \wedge (\forall i = 1..k-1, \mathtt{C_NODES_RSG}(n_i, n_{i+1}) = 1))\}.$$

• The new set of pvar references, $PL(rsg_c)$, is basically the same set of the uncompressed RSG, $PL(rsg)$, but which maps all the nodes into the new ones. This is done with the n_c =$\mathtt{MAP_RSG}(n)$ function which maps the old node $n \in N(rsg)$ into the new node $n_c \in N(rsg_c)$:

$$PL(rsg_c) = \{< pvar, \mathtt{MAP_RSG}(n) >, \forall < pvar, n > \in PL(rsg)\}$$

• The same idea applies to the set of references for the compressed graph:

$$NL(rsg_c) = \{< \mathtt{MAP_RSG}(n_i), sel, \mathtt{MAP_RSG}(n_j) >, \forall < n_i, sel, n_j > \in NL(rsg)\}$$

Regarding the MERGE_COMP_NODES function used for the summarization of several compatible nodes, we define:

$$\mathtt{MERGE_COMP_NODES}(n_1, ...n_k) = \mathtt{MERGE_NODES}(n_1, \mathtt{MERGE_NODES}(n_2, ..., \mathtt{MERGE_NODES}(n_{k-1}, n_k)...))$$

The $\mathtt{MERGE_NODES}(n_1, n_2)$ function takes care of the summarization of nodes n_1 and n_2 into node n. The properties of this new node, n, are set in order to preserve the description of the data structures represented by the original nodes. This way, $\mathtt{MERGE_NODES}(n_1, n_2) = n$, where:

•$\mathtt{SELINset}(n) = \mathtt{SELINset}(n_1) \cap \mathtt{SELINset}(n_2)$
•$\mathtt{SELOUTset}(n) = \mathtt{SELOUTset}(n_1) \cap \mathtt{SELOUTset}(n_2)$
•$\mathtt{PosSELINset}(n) = (\mathtt{SELINset}(n_1) \cup \mathtt{SELINset}(n_2) \cup \mathtt{PosSELINset}(n_1) \cup \mathtt{PosSELINset}(n_2)) \setminus \mathtt{SELINset}(n)$
•$\mathtt{PosSELOUTset}(n) = (\mathtt{SELOUTset}(n_1) \cup \mathtt{SELOUTset}(n_2) \cup \mathtt{PosSELOUTset}(n_1) \cup \mathtt{PosSELOUTset}(n_2)) \setminus \mathtt{SELOUTset}(n)$
•$\mathtt{CYCLELINKS}(n) = \{< sel_i, sel_j > \mid$
$< sel_i, sel_j > \in (\mathtt{CYCLELINKS}(n_1), \mathtt{CYCLELINKS}(n_2)) \vee$
$< sel_i, sel_j > \in \mathtt{CYCLELINKS}(n_1) \wedge \nexists n_k \in N(rsg),$
$< n_2, sel_i, n_k > \in NL(rsg) \vee$
$< sel_i, sel_j > \in \mathtt{CYCLELINKS}(n_2) \wedge \nexists n_k \in N(rsg),$
$< n_1, sel_i, n_k > \in NL(rsg)\}$

The sets SELINset, SELOUTset, PosSELINset and PosSELOUTset keep the *reference pattern* information of nodes (see [2]). The new node n will have the same TYPE, STRUCTURE, SHARED, SHSEL, and TOUCH properties which actually should be the same in n_1 and n_2 to allow the summarization of both nodes. However, the new reference pattern information behaves conservatively. If sel_i is or it is not an input/output selector in both nodes, n_1 and n_2, then it will remain the same in n. In other case, sel_i becomes a *possible* input/output selector.

Finally, regarding the CYCLELINKS sets, the resulting node n keeps the common cycle links sets from the original nodes, n_1 and n_2. In addition, a cycle link, $< sel_i, sel_j >$, from one of the nodes, for example, n_1, is also included in the cycle link set of the node n if the first selector sel_i is not a link selector in the other node, n_2.

4. Reduced Set of RSGs

We have already seen that an RSG describes a memory configuration by a finite graph. Due to the control flow of the program, the same sentence could be reached by several control paths leading to several memory configurations. 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).

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. Thus, we define $\text{COMPATIBLE}(rsg_1, rsg_2) = \text{true}$ if $\text{ALIAS}(rsg_1) = \text{ALIAS}(rsg_2)) \wedge \text{COMP_NODES}(rsg_1, rsg_2) = 1$. We see that two graphs, rsg_1 and rsg_2, are compatible if 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:

- $\text{ALIAS}(rsg)$ 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:

$\text{ALIAS}(rsg) = \{alr_1, ..., alr_n\}$ where $alr_i = \{pv_1, .., pv_m \in P\}$ if $\exists n_i \in N(rsg) | < pv_1, n_i >, ..., < pv_m, n_i > \in PL(rsg)$

- $\text{COMP_NODES}(rsg_1, rsg_2)$ is a Boolean function which returns true if the nodes directly pointed to by the same pvar are compatible. This function is formalized in two steps: the first one identifies the nodes from both RSGs, $n_j \in N(rsg_1)$ and $n_k \in N(rsg_2)$, which are pointed to by the same pvar; the second step determines whether these nodes have similar properties using an additional Boolean function C_NODES:

$\text{COMP_NODES}(rsg_1, rsg_2) = true$ if $\forall pvar_i \in P$,
$< pvar_i, n_j > \in PL(rsg_1) \wedge < pvar_i, n_k > \in PL(rsg_2) \wedge$
$\text{C_NODES}(n_j, n_k) = 1$.
Where
$\text{C_NODES}(n_1, n_2) = true$ if $(\text{TYPE}(n_1) = \text{TYPE}(n_2)) \wedge$
$(\text{SHARED}(n_1) = \text{SHARED}(n_2)) \wedge (\text{SHSEL}(n_1, sel_i) =$
$\text{SHSEL}(n_2, sel_i) \forall sel_i \in S) \wedge (\text{TOUCH}(n_1) = \text{TOUCH}(n_2)) \wedge$
$(\text{C_REFPAT}(n_1, n_2) = 1) \wedge (\text{C_SPATH}(n_1, n_2, m) = 1)$

In the following subsections, we describe the operations that can be carried out with the RSGs of an RSRSG, as we have seen in Fig. 2.

4.1. Graph division

The division operation takes place for the $x \to sel = NULL$, $x \to sel = y$ and $y = x \to sel$. The goal of this operation is to split the input rsg into several graphs, such that for each one of these graphs, the node directly pointed to by x points to a single node by selector sel. Going back to Fig. 1 (a), we can see how the rsg is divided into graphs $rsg\prime_1$ and $rsg\prime_2$ (Fig. 1 (b)), in such a way that in each resulting graph, n_1 points to a single node by nxt.

Basically, with the division, we try to recover the individual characteristics of memory configurations that were fused into a single RSG in a previous sentence. However, this division can generate redundant or inexistent nodes, and links which should be removed by the subsequent PRUNE operation, achieving a more precise representation.

We define $\text{DIVIDE}(rsg, x, sel) = \{rsg_1, .., rsg_n\}$ which divides the rsg in the set $\{rsg_1, .., rsg_n\}$ regarding the pvar x and selector sel. This division is carried out in the following way. If $n \in N(rsg) | < x, n > \in PL(rsg)$, then, $\forall < n, sel, n_i > \in NL(rsg)$, we create a $rsg\prime_i$ such that $N(rsg\prime_i) = N(rsg)$, $PL(rsg\prime_i) = PL(rsg)$ and $NL(rsg\prime_i) = NL(rsg) \setminus \{< n, sel, n_j > \in NL(rsg), \forall n_j \neq n_i\}$. Each $rsg\prime_i$ can contain a single node n_i pointed to by n by selector sel. This $rsg\prime_i$ is subsequently pruned to obtain the definitive $rsg_i = \text{PRUNE}(rsg\prime_i)$ for the DIVIDE function. This pruning process is described next.

4.2. Graph pruning

After graph division, there can be nodes or links in a graph which are not compliant with the new properties of this graph. These nodes or links can be removed because they belong to other graph resulting from the division operation.

In our previous example we can see how the two divided graphs of Fig. 1(b) are simplified into the graphs presented in Fig. 1(c). Note that $rsg\prime\prime_1$ is obtained after the

pruning of $rsg\prime_1$, in which we can safely remove the link $< n_3, prv, n_1 >$ due to it not fulfilling the cycle link properties of node n_3. This property states that subsequently following prv and nxt from node n_3, this n_3 should be reached, but this does not happen for the above-mentioned link. Regarding $rsg\prime\prime_2$, note that the same happens for the link $< n_2, prv, n_1 >$. Besides this, because node n_3 is *not shared* by selector nxt and we are sure that $< n_1, nxt, n_3 >$ exists, we can conclude that $< n_2, nxt, n_3 >$ should be removed. This implies the elimination of $< n_3, prv, n_2 >$ due to cycle link properties. After this elimination, node n_2 cannot be reached and is therefore removed from $rsg\prime\prime_2$. It is important to note that the false value in share attributes leads to a more aggressive pruning which simplifies the RSRSGs and greatly contributes to avoid an explosion in the number of nodes.

To formalize the pruning process, our method uses two Boolean functions to determine whether a node, $\texttt{N_PRUNE}(n)$, or a link, $\texttt{NL_PRUNE}(< n_1, sel, n_2 >)$, fulfill the graph properties:

• A certain node, n, is removed from the graph only taking into account the reference pattern property. That is, if the *reference patterns sets* assert that the node is referenced by selector sel or that this node references by sel to another node, these conditions should be hold. In other cases, the node should be eliminated from the graph. More formally

$\texttt{N_PRUNE}(n) = true$ if $(\exists sel_i \in \texttt{SELOUTset}(n) \wedge sel_i \notin \texttt{posSELOUTset}(n) \wedge \nexists n_2 \in N(rsg), < n, sel_i, n_2 > \in NL(rsg)) \vee (\exists sel_i \in \texttt{SELINset}(n) \wedge sel_i \notin \texttt{posSELINset}(n) \wedge \nexists n_2 \in N(rsg), < n_2, sel_i, n > \in NL(rsg))$

• On the other hand, the link restrictions arise due to the CYCLELINKS property. For example, let's assume a certain node, n, has a set of CYCLELINKS which comprises this particular one: $< sel_1, sel_2 >$. This cycle link points out that the memory locations represented by the node points to others by selector sel_1, and those ones point to the original locations by selector sel_2. Therefore, in our example, if the node n_2 pointed to by n_1 does not point again to n_1 using selectors sel_1 and sel_2, respectively, we can safely remove the link $< n_1, sel_1, n_2 >$. Formally

$\texttt{NL_PRUNE}(< n_1, sel_i, n_2 >) = true$ if $\exists < sel_i, sel_j > \in \texttt{CYCLELINKS}(n_1) | < n_2, sel_j, n_1 > \notin NL(rsg)$

Additionally, we should note that this pruning is an iterative process, because after the elimination of some nodes or links there may appear more nodes or links which do not fulfill the rules and should be removed too. This way, the pruning process ends when all the nodes and links fulfill the graph properties. The whole process can be expressed as $\texttt{PRUNE}(rsg) = rsg_n$. This iterative function starts with $rsg_0 = rsg$. Next, $\forall i = 1..n$:

$N(rsg_i) = N(rsg_{i-1}) \setminus \{n \in N(rsg_{i-1}) | \texttt{N_PRUNE}(n) = 1\}$,
$PL(rsg_i) = PL(rsg_{i-1}) \setminus \{< pvar, n > \in PL(rsg_{i-1}) | \texttt{N_PRUNE}(n) = 1\}$ and
$NL(rsg_i) = NL(rsg_{i-1}) \setminus \{< n_1, sel, n_2 > \in NL(rsg_{i-1}) | (\texttt{N_PRUNE}(n_1) = 1) \vee (\texttt{N_PRUNE}(n_2) = 1) \vee (\texttt{NL_PRUNE}(< n_1, sel, n_2 >) = 1)\}$

where for each iteration we remove the nodes and links for which functions N_PRUNE and NL_PRUNE return true. The process ends when we reach the graph rsg_n which holds $\forall n \in N(rsg_n), \texttt{N_PRUNE}(n) = 0 \wedge \forall < n_1, sel, n_2 > \in NL(rsg_n), \texttt{NL_PRUNE}(< n_1, sel, n_2 >) = 0$

4.3. Graph union

To obtain the RSRSG for a sentence, two compatible graphs rsg_1 and rsg_2, ($\texttt{COMPATIBLE}$ $(rsg_1, rsg_2) = 1$), can be fused in a single graph, rsg, which captures the data structure information represented by the two original graphs. This union of graphs is carried out by the $\texttt{JOIN}(rsg_1, rsg_2) = rsg$ function. Some of the nodes of rsg_1 and rsg_2 are going to be summarized if they are compatible. Now, using the function MERGE_NODES described in Sect. 3.1 we can describe the sets N, PL, and NL of the new rsg, resulting from the union of rsg_1 and rsg_2:

• The set of nodes, N, for the new graph, rsg, comprises three subsets: the non-compatible nodes from rsg_1, the non-compatible nodes from rsg_2, and the nodes resulting from the union of compatible nodes (MERGE_NODES):

$N(rsg) = \{n_i \in N(rsg_1) | \nexists n_j \in N(rsg_2), \texttt{C_NODES}(n_i, n_j) = 1)\} \cup \{n_i \in N(rsg_2) | \nexists n_j \in N(rsg_1), \texttt{C_NODES}(n_i, n_j) = 1)\} \cup \{n = \texttt{MERGE_NODES}(n_i, n_j), \forall n_i \in N(rsg_1), \forall n_j \in N(rsg_2) | (\texttt{C_NODES}(n_i, n_j) = 1)\}$

We use a $\texttt{MAP}(n_i) = n$ function to describe the new $PL(rsg)$ and $NL(rsg)$ sets. This function points out which node n of the new graph is now representing the node n_i from rsg_1 or rsg_2.

• The set of references from pvars to nodes $PL(rsg)$ are obtained by translating the old references from rsg_1 and rsg_2 to the new graph using the MAP function.

$PL(rsg) = \{< pv, MAP(n_i) > | \forall (< pv, n_i > \in PL(rsg_1)\} \cup \{< pv, MAP(n_j) > | \forall (< pv, n_j > \in PL(rsg_2)\}$

• Similarly, we obtain the set of links between nodes:

$NL(rsg) = \{< MAP(n_i), sel_j, MAP(n_k) > | \forall < n_i, sel_j, n_k > \in NL(rsg_1)\} \cup \{< MAP(n_i), sel_j, MAP(n_k) > | \forall < n_i, sel_j, n_k > \in NL(rsg_2)\}$

This way, in the new graph, rsg, we keep all the references and links existing in the original graphs, rsg_1 and rsg_2, just changing the source and destination nodes.

5. Experimental results

All the previously described 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 have seen, the set of properties associated with a node allows the compiler to keep in separate nodes those memory locations with different properties. Obviously, the number of nodes in the RSRSGs depends on the number of

	S.Mat-Vec	S.Mat-Mat
Level	L_1 / L_2 / L_3	L_1 / L_2 / L_3
Time	0'03"/0'05"/0'07"	0'51"/1'36"/1'57"
Space (MB)	1.37/1.85/2.17	8.13/11.45/12.68
	S.LU fact.	**Barnes-Hut**
Level	L_1	L_1 / L_2 / L_3
Time	12'15"	17'01"/1'47"/3'21"
Space (MB)	99.46	44.46/12.44/21.31

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

properties and also on the range of values these properties can take. The higher the number of properties the better the accuracy in the memory configuration representation, but also the larger the RSRSGs and memory wastage. Fortunately, not all the properties are needed to achieve a precise description of the data structure in all the codes.

Bearing this in mind, we have implemented the compiler to carry out a **progressive analysis** which starts with fewer constraints to summarize nodes, but, when necessary, these constraints are increased to reach a better approximation of the data structure used in the code. More precisely, the compiler analysis comprises three levels:

• L_1: In this level the TOUCH sets are not built nor taken into account and only the C_SPATH0 condition is used.

• L_2: This level is based on the previous one but now using the C_SPATH1.

• L_3: This is the higher level in which all the properties including the TOUCH one are taken into account.

With this compiler we have analyzed several C codes: the *sparse Matrix by vector multiplication*, the *sparse Matrix by Matrix multiplication*, the *Sparse LU factorization*, and the *Barnes-Hut* code. The first three codes were successfully analyzed in the first level of the compiler, L_1 (a detailed description of these codes and the compiler results is in [2]). However, for the Barnes-Hut program the highest accuracy of the RSRSGs was obtained in the last level, L_3, as we explain in Sect. 5.1. All these codes where analyzed by our compiler in a Pentium III 500 MHZ with 128 MB main memory. The time and memory required by the compiler are summarized in Table 1.

As we mentioned, all sparse codes are accurately analyzed in the compiler L_1 level. However, in Table 1 we also present time and memory requirements of levels L_2 and L_3 to illustrate how increasing the number of properties usually leads to higher compilation times and larger memory pools. For the Sparse LU factorization, our compiler runs out of memory in L_2 and L_3 in our 128 MB Pentium III and, therefore, only values for L_1 level are provided. The exception in Table 1 is due to the Barnes-Hut N-body simulation which is described next.

5.1. Barnes-Hut N-body simulation

This code deserves more attention due to it being a good example in which the compiler has to sequentially carry out the three levels of compilation in the progressive analysis.

This code is a classical gravitational N-body simulation which simulates the evolution of a system of bodies under the influence of gravitational forces. The data structure used in this code is based on a hierarchical octree representation of space in three dimensions. In Fig. 3(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 $Lbodies$. The octree represents the several subdivisions of the 3D space.

The three main steps in the algorithm are: (i) The creation of the octree; (ii) for each subsquare in the octree, compute the center of mass and total mass for all the particles it contains, traversing the tree; and (iii) for each particle, traverse the tree to compute the forces on it.

All the traversals of the octree are carried out in the code in recursive calls. Due to the fact that our compiler is still not able to perform an interprocedural analysis, we have manually carried out the inline of the subroutine and the recursivity has been transformed into a loop. This loop uses a stack pointing to the nodes which are referenced during the octree traversal. This stack is also considered in Fig. 3 (a) and obtained in the corresponding RSRSG, Fig. 3 (b).

After the L_1 analysis of the code, the resulting RSRSG for the last sentence of the code does not correspond with the real properties of the data structure used in the code. The problem is that the summary node n_6, which represents the middle elements of the $Lbodies$ list fulfill SHSEL(body) = true. This would imply that there may be different leaves in the octree pointing to the same body in the $Lbodies$ list. This inaccuracy is due to an imprecise analysis during the generation of the list of children in the octree, and it is avoided moving on to the L_2 level, thanks to the use of C_SPATH1. The resulting RSRSG can be seen in Fig. 3(b), where the summary node n_6 fulfills $\mathtt{SHSEL}(n_6, body)$ = false, in line with the real data structure. However, due to the stack we use to assist the octree traversal, there is a problem that cannot be solved in L_2: nodes n_2, n_3, and n_4 are shared by selector $node$ from the $Stack$ data structure. This would prevent the parallel traversal of the octree in steps (ii) and (iii).

However, by switching to the L_3 compilation level this problem is solved for traversal (iii) thanks to the TOUCH property. A subsequent analysis of the code can state that the tree can be traversed and updated in parallel on step (iii).

However, regarding the Table 1, there is a paradoxical behavior that deserves an explanation: L_2 and L_3 expend

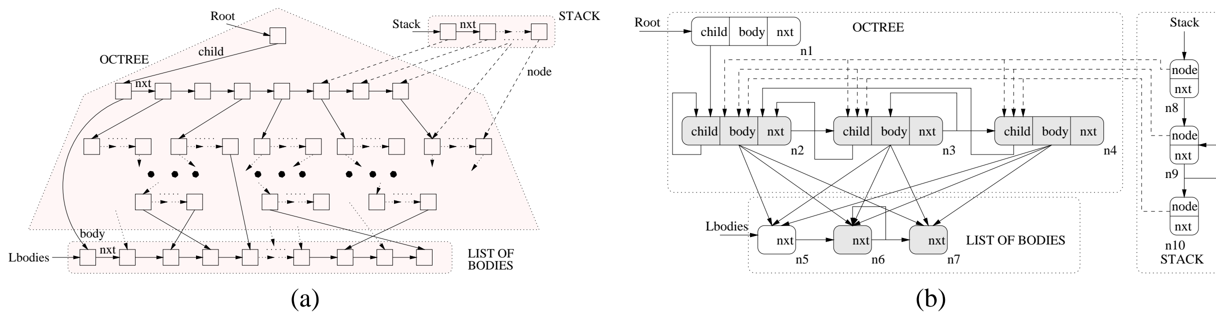

Figure 3. Barnes-Hut data structure and compacted RSRSG.

less time and memory than L_1. As was briefly described in the example in Sect. 4.2, when SHARED and SHSEL are false there are more nodes and links pruned during the abstract interpretation of a sentence. In this code, for the L_2 and L_3 levels, the $\mathtt{SHSEL}(n_6, body)$ = false leads to more pruning and graph simplifications counteracting the increase in complexity.

6. 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 and complexity of properties assigned to each node. This leads to more nodes in the RSG, however, we keep a more accurate representation of the data structure. This is a key issue when analyzing the parallelism exhibited by a code. Besides, the compiler carries out a progressive analysis, increasing the complexity of the compilation process only for those codes which really need it.

We have validated the compiler with several C codes which generate, traverse, and modify complex dynamic data structures.

In the near future we want to face the interprocedural analysis problem to deal with recursive calls. We will also develop an additional compiler pass that will automatically analyze the RSRSGs and the code to determine the parallel loops and allow the automatic generation of parallel code.

References

[1] 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.

[2] F. Corbera, R. Asenjo and E.L. Zapata. *Accurate Shape Analysis for Recursive Data Structures.* In 13th Int'l. Workshop on Languages and Compilers for Parallel Computing, IBM T.J. Watson Res. Ctr., New York, August 2000.

[3] E. Gutirrez, R. Asenjo, O. Plata and E.L. Zapata. *Automatic Parallelization of Irregular Applications.* J. Parallel Computing, vol. 26, no. 13-14, December 2000, pp. 1709-1738.

[4] 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.

[5] 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.

[6] J. Plevyak, A. Chien and V. Karamcheti. *Analysis of Dynamic Structures for Efficient Parallel Execution.* In Languages and Compilers for Parallel Computing, Eds. Lectures Notes in Computer Science, vol 768, 37-57. Berlin Heidelberg New York: Springer-Verlag 1993.

[7] 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.

[8] M. Sagiv, T. Reps, and R. Wilhelm. *Parametric shape analysis via 3-valued logic*. In Conf. Record of the 26th ACM Symposium on Principles of Programming Languages, San Antonio, TX, ACM, NY, Jan. 1999, pp. 105-118.

[9] Y. Hwang and J. Saltz. *Identifying DEF/USE information of statements that construct and traverse dynamic recursive data structures* In Proceedings of 10th International Workshop on Languages and Compilers for Parallel Computing, University of Minnesota, August 1997.