Towards Compiler Optimization of Codes Based on Arrays of Pointers

F. Corbera
R. Asenjo
E.L. Zapata

July 2002
Technical Report No: UMA-DAC-02/13

Published in:

15th Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC'02)
College Park, Maryland, July 25-27, 2002

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

Towards Compiler Optimization of Codes Based on Arrays of Pointers

F. Corbera[1], R. Asenjo[1], and E.L.Zapata[1]

Computer Architecture Dpt., University of Malaga
{corbera, asenjo, ezapata}@ac.uma.es

Abstract. To successfully exploit all the possibilities of current computer/multicomputer architectures, optimization compiling techniques are a must. However, for codes based on pointers and dynamic data structures these optimization techniques have to be necessarily carried out after identifying the characteristics and properties of the data structure used in the code. In this paper we present one method able to automatically identify complex dynamic data structures used in a code even in the presence of arrays of pointers. This method has been implemented in an analyzer which symbolically executes the input code to generate a set of graphs, called RSRSG (Reduced Set of Reference Shape Graphs), for each statement. Each RSRSG accurately describes the data structure configuration at each program point. In order to deal with arrays of pointers we have introduced two main concepts: the multireference class, and instances. Our analyzer has been validated with several codes based on complex data structures containing arrays of pointers which were successfully identified.

1 Introduction

Programming languages such as C, C++, Fortran90, or Java are widely used for non-numerical (symbolic) and numerical applications. All these languages allow the use of complex data structures usually based on pointers and dynamic memory allocation. The use of complex data structures is very helpful in order to speedup code development and, besides this, it also may lead to reducing the program execution time. However, compilers are not able to successfully optimize codes based on these complex data structures for current computers or multicomputers.

More precisely, when dealing with pointer-based data structures usually built at run time, current compilers are not able to capture, from the code text, the necessary information to exploit locality, automatically parallelize the code, or carry out other important optimizations. In other words, if the compiler is not aware of the properties fulfilled by the data structure used in the code, it is impossible to apply certain optimizations. For instance, if the compiler does not know that a certain loop is traversing a doubly linked list, then important techniques such as data prefetching, locality exploiting or parallelism detection, cannot be applied.

With this motivation, the goal of our research line is to propose and implement new techniques that can be included in compilers to allow for the automatic optimization of real codes based on dynamic data structures. As 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, subsequent analysis (not implemented yet) would focus on particular optimizations, for example, to exploit the memory hierarchy or to detect whether or not certain sections of the code can be parallelized because they access independent data regions.

There are several ways this shape analysis problem can be approached, some of which are based on explicit programmer annotations [9], and others are based on abstracting the properties of data structures by means of "path expressions" [11], "matrices" [6] or graphs. We have focussed on graph-based methods as they are able to explicitly keep information about dynamic objects not pointed to by any pointer variable. In these graphs the nodes represent the "storage chunks" and edges are used to represent references between them. Some of these graph-based methods use just one graph to approximate all possible memory configurations for each statement in the code [2, 12, 13], whereas other methods permit the existence of several graphs per statement to represent the information more accurately [10, 8, 14]. Our own method belongs to the later class, and it is described in [4, 5]. Basically, our analyzer generates a reduced set of reference shape graphs (RSRSG) for each statement in the code. Each RSRSG approximates the data structure at each corresponding program point. We have compared our analyzer with other related works in [4, 5], but we emphasize here that to the best of our knowledge, our analyzer is the only one able to accurately identify the data structure at each statement of a real C code. The analyzed codes are based on complex data structures such as doubly linked lists, trees, and octrees among others, and combinations of them, such as a doubly linked list of pointers to trees where the leaves point to doubly linked lists, etc.

However, our analyzer was not able to handle arrays of pointers as part of the dynamic data structure. Therefore, in this paper we describe how we extend our method to deal with structures containing arrays of pointers. Please, note that this is the main goal of this paper and that the analyzer details (which are already covered in [4, 5]) can not be tackled here due to space constrainst. Again, as far as we know, there is no other previous technique able to automatically identify complex data structures comprising arrays of pointers. However, arrays of pointers are frequently included in the definition of complex and dynamic data structures such as sparse matrices and quad/octrees, as we see in Sect. 5, and therefore they deserve to be taken into account in the area of shape analysis research.

The rest of the paper is organized as follows. In Sect. 2 we provide an overview of our shape analysis method, briefly summarizing our previous work for the sake of completeness, as the next sections are based on these ideas. Sect. 3 introduces new contributions to deal with arrays of pointers, such as the multiselector and instance ideas. These new issues lead to new steps in shape analysis which are described in Sect. 4. In Sect. 5 we present the experimental results obtained after feeding our analyzer with several codes based on structures comprising arrays of pointers. Finally, we conclude with the main contributions and future work in Sect. 6.

2 Method Overview

Basically, our method presented in [4, 5] is based on approximating all possible memory configurations that can appear after the execution of a statement in the code. 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*. Each statement in the code will have a set of *Reference Shape Graphs* (RSG) associated with it, which are called a *Reduced Set of Reference Shape Graphs* (RSRSG). The RSGs are graphs in which nodes represent memory locations which have similar reference patterns. 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, if several memory locations share the same properties, then all of them will be represented by the same node. These properties are described in [4, 5], but to understand the experimental results we have to explain one of them: the share information. This property can tell whether at least one of the locations represented by a node is referenced more than once from other memory locations. In order to hold the shared information we use two kinds of attributes for each node: *SHARED(n)* states if any of the locations represented by the node n can be referenced by other locations by different selectors, and *SHSEL(n, sel)* points out if any of the locations represented by n can be referenced more than once by following the same selector sel from other locations.

As we have said, all possible memory configurations which may arise after the execution of a statement are approximated by a set of RSGs we call RSRSG. To move from the “memory domain” to the “graph domain”, the calculation of the RSRSGs associated with a statement is carried out by the **symbolic execution** of the program over the graphs. In this way, each program statement transforms the graphs to reflect the changes in memory configurations derived from statement execution. The **abstract semantic** of each statement states how the analysis of this statement must transform the graphs. The whole symbolic execution process can be seen by looking at Fig. 1. For each statement in the code we have an input $RSRSG_i$ and the corresponding output $RSRSG_o$ representing the memory configurations after statement execution. During the symbolic execution of the statement all the rsg_{ij} belonging to $RSRSG_i$ are going to be updated. The first step comprises graph division to better focus on the several memory configurations represented by the RSG. Pruning removes redundant or nonexistent nodes or links that may appear after the division operation. Then the abstract interpretation of the statement takes place and usually the complexity of the RSGs grows. In order to counter this effect, the analysis carries out a compression operation. In this phase each RSG is simplified by the summarization of compatible nodes, to obtain the rsg^*_{ijk} graphs. Furthermore, some of the rsg^*_{ijk} 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.

In the next two sections we present the new extension that allows our analyzer to deal with dynamic data structures comprising arrays of pointers, as these kinds of data structures are widely used in C codes. Due to space constraints we have tried to present

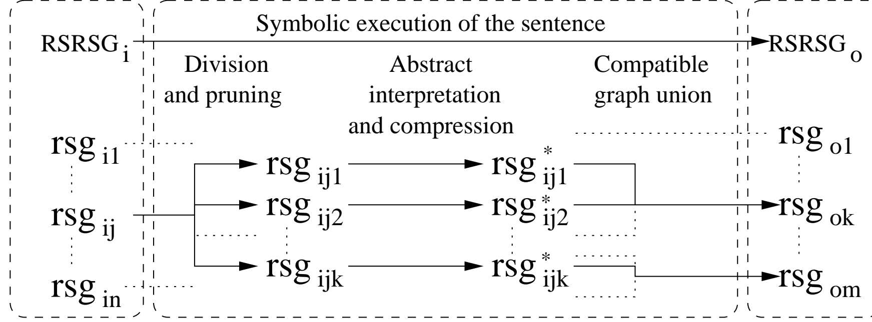


Fig. 1. Schematic description of the symbolic execution of a statement.

a clear idea of our extensions using English and examples, but more technical details are in [3].

3 Multiselectors

We can view an array of pointers as a set of n selectors (links), all with the same name. Our original method, briefly described in the previous section, only deals with single selectors (which represent single links). Thus, the problem arising with the arrays of pointers is that a single selector name represents several links, and all of them belong to the same memory location (due to having been allocated by the same *malloc* instruction).

We illustrate all this with the following example. Figure 2 shows an example of a complex data structure definition comprising two arrays of pointers, and it also illustrates the corresponding memory configuration after the execution of the last "malloc()" statement. As we note, sel is a single selector which can point to a single memory location and which can be modified by statements like "x→sel=...". These kinds of selectors can be managed by our previous analyzer. However, $sel1$ and $sel2$ represent arrays of selectors. The difference between $sel1$ and $sel2$ is that we know the size of the $sel1$ array at compile time, but the size of $sel2$ is defined at run time. In any case, we now want to deal with both types of arrays of selectors, which now have to be modified by statements like "x→sel1[i]=..." or "x→sel2[i]=...".

Since $sel1$ and $sel2$ are not single selectors, we have called them *multiselectors*. In order to take into account multiselectors in our method we have introduced in our analyzer two new important concepts: **instance** and **multireference class**. The idea is the following: since our method is already able to deal with single selectors our goal is now to include a previous step in the symbolic execution process to focus on one of the selectors included in a particular multiselector. In other words, a statement like "x→sel1[i]=..." is going to update a single selector (a particular selector included in the multiselector $sel1$), but before applying the symbolic execution, first we have to identify the particular sel1[i] which is going to be updated. The instances and multireference classes will help us to develop this preprocessing stage.

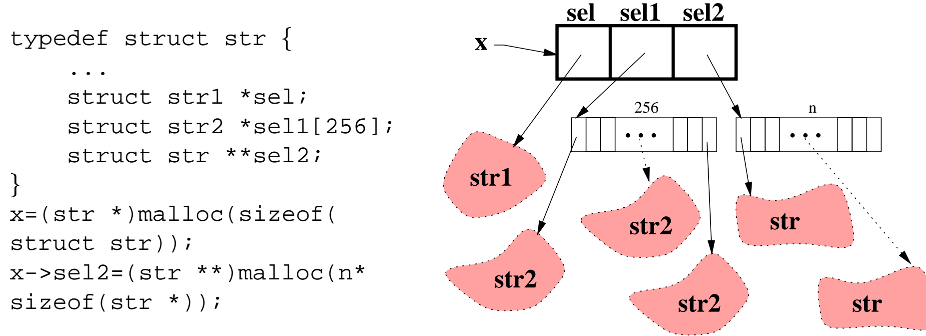


Fig. 2. Example of data structure containing arrays of pointers

3.1 Instances

An instance of a multiselector represents a subset of links belonging to this multiselector. In other words, an instance identifies a subregion in the array of pointers. For example, for the statement $x \rightarrow sel[i]$ the analyzer creates an instance in the multiselector sel (the one directly pointed to by x), which represents the i position of array sel. This way, this instance can be processed and modified by the analyzer as if it were a single selector.

In our method, the set of variables which are used to index the arrays of pointers is called *IVARS*. Now, an instance, ins, is identified by two sets, $< ivs, vivs >$ where:

• $ivs = \{iv \in \mathit{IVARS}\}$, is the set of index variables which identifies the array position represented by the instance.

• $vivs = \{iv \in \mathit{IVARS}\}$, is the set of index variables that have previously visited the array position represented by the instance, but which are currently indexing other array positions.

The reason to keep the $vivs$ set, is to achieve a more accurate description and processing of the link represented by $x \rightarrow sel[i]$ inside a loop body in which the variable i does not take the same value twice. Therefore, the reference $x \rightarrow sel[i]$ always identifies a different position of the array sel and the analyzer will be able to avoid updating regions of the data structures already updated in a previous iteration of the loop.

The functions $ivs(ins)$ and $vivs(ins)$ provide the ivs and $vivs$ sets respectively. Now, we can say that there are two types of instances:

• *Single instance*. These instances represent exactly one position of the array and they can be handled as a single selector. The instance ins is a *single instance* if $ivs(ins) \neq \emptyset$, which means that there is an index variable in the ivs set for the corresponding ins instance.

• *Multiple instance*. These instances represent more than one array position, i.e. one array region. Now, if ins is a *multiple instance*, then $ivs(ins) = \emptyset$.

We illustrate all these concepts with an example. In Fig. 3 we can see a graph associated with the statement 4 of the code presented in the same figure. Actually, at this program point, the pointer variable (pvar) $tree$ is pointing to the root of the tree repre-

sented by node $n1$. This tree root contains an array of pointers to the children, called $child$. Therefore, the node $n1$ contains the $child$ multiselector. In this example, for this multiselector, there are three instances represented by three circular nodes.

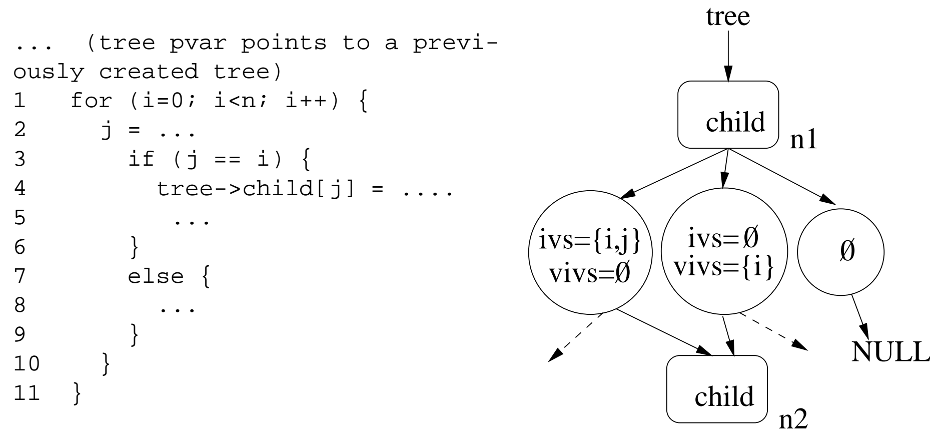


Fig. 3. Code example and graph representation of the instances.

If statement 4 is reached, clearly index variables i and j share the same value and the corresponding RSRSG will reflect this fact. Actually, the first instance, identified by $< \{i, j\}, \emptyset >$ (which means $ivs = \{i, j\}, vivs = \emptyset$), represents the single array position indexed by variables i or j. Clearly, this is a single instance as index variables i or j have a single value at this program point and this way they index a single array position. The $vivs = \emptyset$ for this instance states that this array position was not previously visited by any index variable.

The second instance is identified by $< \emptyset, \{i\} >$. This is a multiple instance representing all the array positions not indexed at the current statement by any index variable but previously visited by the index variable i in previous iterations of the loop at statement 1. Finally, the last instance, identified by $\emptyset$ ($< \emptyset, \emptyset >$), represents all the other array positions: not indexed now or before by index variable i.

For example, if $i = 5$, at statement 4, the instance $< \{i, j\}, \emptyset >$ represents the position 5 of the array, the instance $< \emptyset, \{i\} >$ represents positions 0 to 4 and the instance $\emptyset$ identifies positions 6 to the final one.

3.2 Multireference Classes

As we saw in Sect. 2, our method symbolically executes each statement of the code, and some of the operations included in the symbolic execution are graph compression and union and graph division. Graph compression and union operations are necessary to avoid an explosion in the number of nodes and graphs associated with each statement to describe the data structure at each program point. On the other hand, the goal of the

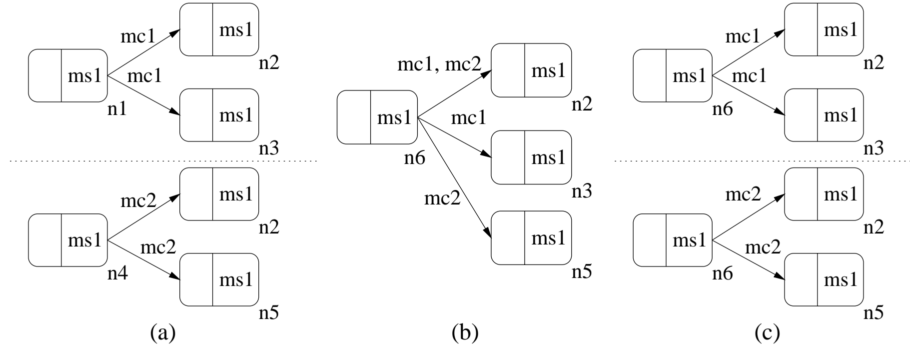


Fig. 4. Compression and division with multireference classes

division operation is to focus on the area of the graph which is going to be modified by the symbolic execution of a statement, leading to a much more accurate updating of the RSRSGs. These operations were defined and work well for single selectors, but something new has to be introduced to also deal with multiselectors: the multireference classes, (mc).

After the allocation of a new memory location, multiselectors are labeled with a certain multireference class. During the graph compression and union operations, compatible nodes which represent similar memory locations are fused or *summarized* into a single one. When two nodes are summarized the destinations of their *selectors* and *multiselectors* may be joined as well, but multiselector preserves their multireference class. Thus, if for a later statement the analyzer wants to focus on one of the summarized nodes, it is possible to separate them, thanks to the multireference classes.

We can better illustrate this with the example in Fig. 4. First we note in Fig. 4 (a) that nodes n_1 and n_4 have the same multiselector ms_1, but links from node n_1 are labeled with mc_1 and those from n_4 belong to the multireference class mc_2. Let's suppose that nodes n_1 and n_4 are compatible and can be summarized into a new node n_6 in Fig. 4 (b). Some of the destination nodes are also joined but the multireference classes allow the analyzer to accurately focus on node n_1 or n_4 if they have to be modified later (Fig. 4 (c)).

Having introduced these two key concepts associated with multiselectors we can move on to briefly describe how the symbolic execution of the statements has to be modified to take into account this new information.

4 Extended Symbolic Execution for Multiselectors

As we said in Sect. 2, the symbolic execution of a statement carries out the generation of the output RSRSG$_o$ which captures the modifications, due to this statement, in the data structures represented by the input RSRSG$_i$. Basically, as we explained in Fig. 1, the symbolic execution process first focusses on the section of the graphs which are going to be updated, to subsequently carry out the abstract interpretation of the statement

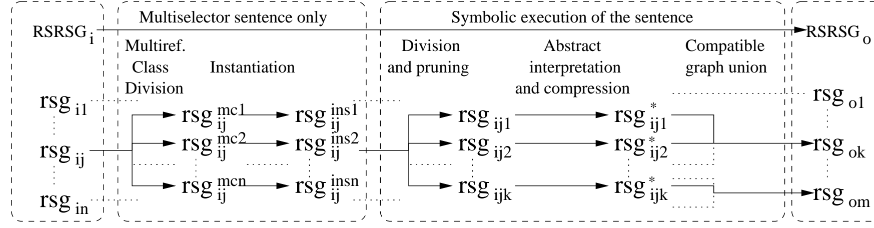


Fig. 5. Extension of the symbolic execution to also deal with multiselectors.

which conveniently modifies the graphs. Finally, graphs are compressed and some of them joined for the sake of memory wastage minimization.

This scheme is valid for statements in which only single selector are involved. However, if the statement includes multiselectors, $x \rightarrow sel[i] = NULL$, $x \rightarrow sel[i] = y$ and $y = x \rightarrow sel[i]$, the analyzer must first identify the i position of the array of pointers to focus on the particular link represented by $sel[i]$. If we are able to do this, then we can later apply the single selector procedure because we have translated a multiselector into a single selector. Since we are carrying out an analysis at compile time, sometimes the method has to behave conservatively: if we cannot identify the particular i selector we have to update all the links that may be represented by the $sel[i]$ selector. Fortunately, the analyzer normally avoids inaccurate updates since it is able to exclude several links that are definitely not represented by $sel[i]$. Basically, in order to focus on a single selector from a multiselector, the analyzer implements two previous steps as we can see in Fig. 5: multireference class division and instantiation. These two preprocessing stages are executed only for the symbolic execution of statements involving multiselector, and are briefly described next.

4.1 Multireference Class Division

For a given statement like $x \rightarrow sel[i]$, the multireference class division operation just splits the different configurations of the links represented by multiselector *sel* into several graphs. These different configurations are in the same graph after a graph union operation and may coexist in the graph domain for the sake of memory saving. However, in the memory domain those configuration are exclusive and the analyzer has to separate them. In other words, in order to increase the accuracy of the method, before updating a graph, the analyzer looks for the most precise description of the memory configurations which are going to be updated.

More precisely, given an rsg_i to be updated by a statement, the multireference class division will split the rsg_i into as many graphs $rsg_i^{mc_i}$ as there are multireference classes in the multiselector, as we can see in Fig. 5. Note that the number of multireference classes that may appear in a multiselector is limited as a multireference class is just an identifier of a subset of links that may be represented by a multiselector. Since the number of links represented by a multiselector is finite (due to there being a finite number of nodes), the number of subsets of links is also finite.

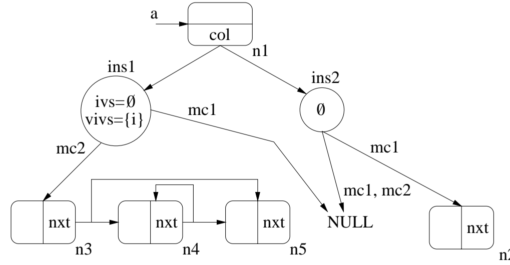


Fig. 6. An example of graph that has to be updated.

This operation can be better illustrated by the example in Fig. 6 where we can see a hypothetical rsg_i before the execution of statement of the type "$a \rightarrow col[i] = ...$". In this graph, the pointer variable a is pointing to a memory location containing the multiselector col. There are two instances associated with this multiselector which are identified by ($< \emptyset, \{i\} >$), instance ins_1, and $\emptyset$, instance ins_2. Both instances are pointing to other locations and the links are labeled with two multireference classes, mc_1 and mc_2.

Put simply, this graph represents an array of pointers, a, where the already visited positions (instance ins_1) are pointing to NULL, mc_1, or to single linked lists of two or more elements (n_3, n_4, and n_5), mc_2, depending of the followed path reaching the statement in the control flow graph. On the other hand, non-visited positions of array a (instance ins_2) may point to a single memory location n_2, mc_1, or to NULL, mc_1, mc_2.

The multireference class division operation generates the two different graphs we can see in Fig. 7. The first graph, Fig. 7 (a), is obtained just by keeping the links belonging to multireference class mc_1, whereas the second graph, Fig. 7 (b), keeps those links of the multireference class mc_2. Now we have identified two possible memory configurations (two possible data structures) that may reach the statement "$a \rightarrow col[i] = ...$". Note that in this example, these two memory configurations reach the same statement after following two different paths in the control flow graph of the analyzed code. The analyzer has to conservatively update each memory configuration according to the new "$a \rightarrow col[i] = ...$" statement because at compile time the analyzer does not know which path is going to be the one executed at run time.

4.2 Instantiation

After the multireference class division, several graphs $rsg_i^{mc_i}$ are going to be modified by the instantiation operation. The goal now is to focus on the particular i position of the array of pointers to successfully translate a multiselector into a single selector. In order to do this, the analyzer has to generate a new single instance to represent the i link of the multiselector sel. This particular link will be later processed as a single selector by the subsequent compiler passes as can be seen in Fig. 5.

More precisely, for a statement of the type $x \rightarrow sel[i]$, the new single instance has to fulfill that $ivs = \{i, ...\}$. In the worst case, the new instance would inherit all the

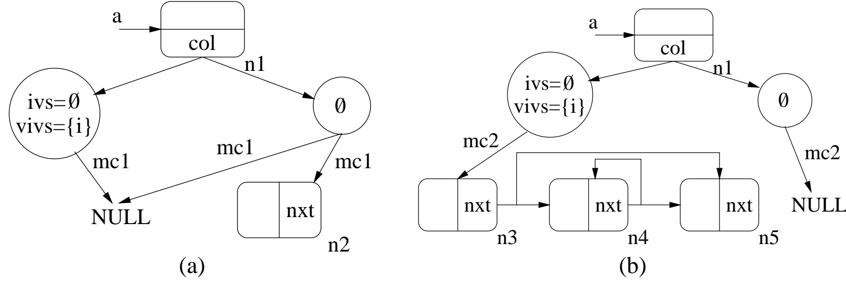


Fig. 7. Graphs obtained after the multireference class division.

links which point to other memory locations from the other already existing instances. This is the most conservative case in which the analyzer is not able to extract more precise information about the particular i position of the array involved in the statement. However, in most cases, the analyzer would be able to identify some relations between index variables. These relations are stored in an *index variable relation table*, *IVRT*, which is going to help in reducing the number of links that the new single instances have to inherit.

This *IVRT*(iv_1, iv_2, st) table has to be generated in a preprocessing compiler pass to store the relations between index variables iv_1 and iv_2 for the statement st. The *IVRT* table also holds the relations between an index variable now (in the current iteration of a loop) and before (in a previous iteration) using the expression *IVRT*$(iv, va(iv), st)$, where $va(iv)$ represents the old values taken by iv in previous iterations of the loop. The possible values for *IVRT*(iv_1, iv_2, st) are: **eq**, if iv_1 and iv_2 have the same value at st; **neq**, if they are different; or **unk**, if the relation between them is unknown. We are also studying including in the $IVRT$ generation pass more precise array region descriptions such as those presented in [7] which also deals with non-affine access functions.

The *IVRT* holds key information regarding the initialization of the links corresponding to the new single instance. This way, the new single instance $< \{i\}, \emptyset >$ has to inherit all the links of the *compatible* instances, which are those that do not contain j in the ivs set where *IVRT*$(i, j, st) = neq$. This is due to the fact that if $i \neq j$ in st, then the instances with $ivs = \{j, ...\}$ do not represent the i position of the array. Besides, if *IVRT*$(i, va(i), st) = neq$, then the new single instance $< \{i\}, \emptyset >$ will not inherit the links of instances of the type $< ..., \{i, ...\} >$.

We can better explain these ideas by reference to Fig. 7. Let's suppose that the analyzer has found out in a previous step that *IVRT*$(i, va(i), st) = neq$, which means that for the code statement st the index variable i has a new value which has never been taken by this variable in this statement st (in a previous iteration). Now, the analyzer has to generate a new single instance $< \{i\}, \emptyset >$ as we see in Fig. 8. Note that this new instance only inherits the links of the $\emptyset$ instance since the $< \emptyset, \{i\} >$ instance identifies already visited positions of the array and we know that i now has a different value.

The number of instances that can appear in any multiselector is limited by the number of index variables and, as we said, an instance is just a pair of sets of index variables.

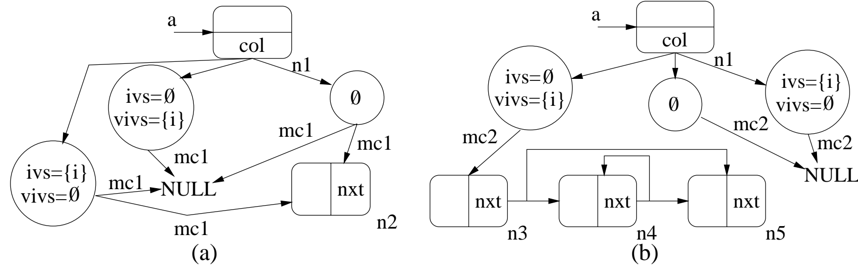


Fig. 8. Resulting graphs after instantiation.

In addition, in these sets they will not appear all possible index variables, but just those involved in the traversing of an array of pointers. These index variables are removed from the instances when the symbolic execution leaves the loop in which the array of pointer is traversed. Subsequently, instances with the same sets are fused and consequently the number of instances decreased.

Due to space constraints we cannot cover additional important issues such as *IVRT* generation and index variable analysis and scope; however, these are described in [3].

5 Experimental Results

With the previously described ideas we have extended the analyzer presented in [4, 5] to allow for the automatic detection of the data structures at each program point for codes based not only on single selectors but also on multiselectors. With this analyzer we have analyzed several codes in which the dominant data structure comprises arrays of pointers.

As we have seen, the set of properties associated with a node allows the analyzer to keep in separate nodes those memory locations with different properties. Obviously, the number of nodes in the RSRSGs depends on the number of 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. That is, simpler codes can be successfully analyzed taking into account fewer properties, and complex programs will need more compilation time and memory due to all the properties that have to be considered to achieve accurate results. Bearing this in mind, we have implemented the analyzer 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, L_2, and L_3, from less to more complexity.

The analyzed codes are the sparse matrix vector multiplication, sparse matrix matrix multiplication, sparse LU factorization, and the kernel of the Barnes-Hut N-body

Table 1. Time and space required by the analyzer to process several codes

	Time	Space (MB)
Level	L_1 / L_2 / L_3	L_1 / L_2 / L_3
S.Mat-Vec	0'03"/0'04"/0'05"	0.92/1.03/1.2
S.Mat-Mat	0'12"/0'14"/0'16	1.19/1.31/1.49
S.LU fact.	2'50"/3'03"/-	3.96/4.18/-
Barnes-Hut	61'24"/69'55"/0'54"	40.14/42.86/3.06

simulation. In Table 1 we present the time and memory required by the analyzer to process these codes in a Pentium III 500 MHZ with 128 MB main memory. The first three codes were successfully analyzed in the first level of the analyzer, L_1. However, for the Barnes-Hut code the highest accuracy of the RSRSGs was obtained in the last level, L_3, as we explain in Sect. 5.2. For the Sparse LU factorization, our analyzer runs out of memory in L_3. We now briefly describe the results for the analyzed codes.

5.1 Sparse Codes

Here we deal with three sparse irregular codes which implement sparse matrix operations: matrix vector multiplication, $r = M \times v$, matrix by matrix multiplication, $A = B \times C$, and sparse LU factorization, $A = LU$.

In the two first codes, sparse matrices M, A, and B are stored in memory as an array of pointers, row, pointing to doubly linked lists which store the matrix rows. Matrix C is similarly stored by columns instead of by rows. The sparse vectors v and r are also doubly linked lists. This can be seen in Fig. 9(a). Note that vector r grows during the matrix vector multiplication process.

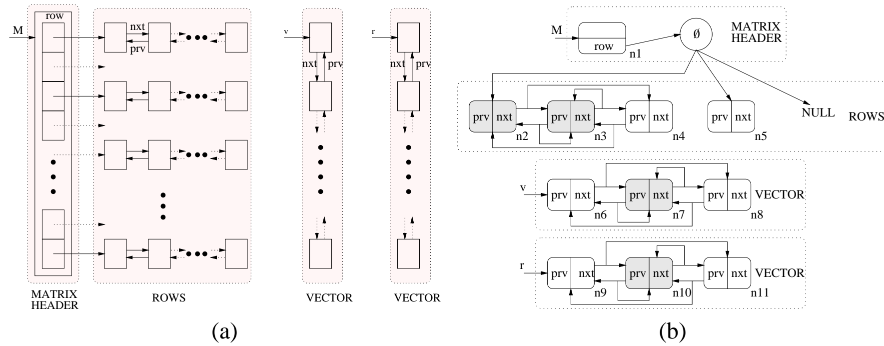


Fig. 9. Sparse matrix-vector multiplication data structure and compacted RSRSG.

On the other hand, the sparse LU factorization solves non-symmetric sparse linear systems by applying the LU factorization of the sparse matrix. Here, the sparse matrix is stored by columns. However, this code is much more complex to analyze due to

the matrix filling, partial pivoting, and column permutation which takes place in the factorization in order to provide numerical stability and preserve the sparseness. After the analysis process, carried out by our analyzer at level L1, the resulting RSRSGs accurately represent the data structure at each program point for the three codes.

Regarding the sparse matrix vector multiplication, in Fig. 9(b) we present a compact representation of the resulting RSRSG for the last statement of the code. Nodes where the *SHARED*(n) property is true are shaded in the figures. In this RSRSG we can clearly see the three main data structures involved in the sparse matrix vector multiplication (M, v, and r). Each vector is represented by three nodes and the central one represents all the middle items of the doubly linked list. The sparse matrix is pointed to by pointer variable M which is actually an array of pointers with the multiselector row. This multiselector has, for the last statement of the code, a single instance ($\emptyset$) representing all the positions (pointers) of the array. In the RSRSG we can see that these pointers can point to *NULL* (there is no element in the row), to a single node (the row has just one entry), or to a doubly linked list of two or more elements. For the matrix matrix multiplication, matrices A, B, and C are also clearly identified by three graphs like the one just described before. The same happens for the in-place sparse LU factorization where the resulting LU matrix is stored where the original matrix A was.

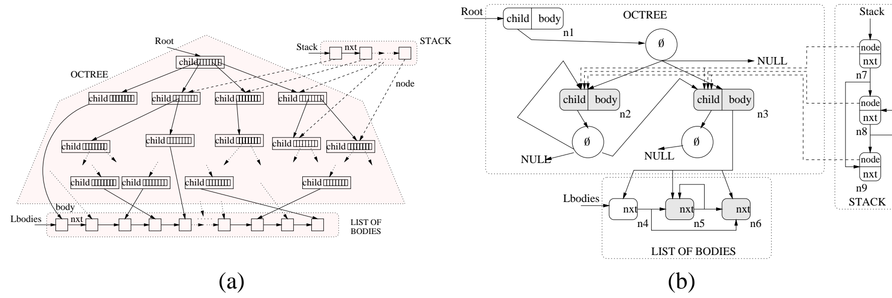
To properly interpret this graph representation of the sparse matrices we have to say that the analyzer also knows that the *SHSEL*(n, sel) for all the nodes and all selectors is false. Remember that *SHSEL*(n, sel) =false means that all the locations represented by n can not be referenced more than once by following the same selector sel from other locations. This leads to several conclusions: (i) the doubly linked lists are acyclic when traversed by following just one kind of selector (nxt or prv), since the *SHSEL*(n, nxt)=false points out that a node can not be pointed to twice by other nodes using selector nxt (the same for prv); (ii) different pointers of the array row point to different rows, as *SHSEL*(n_2, row) = false; (iii) besides this, the doubly linked lists do not share elements between them.

Using this information, a subsequent compiler pass would be able to identify the traversals of the rows for prefetching or locality exploiting. Furthermore, the analyzer would state that the sparse matrix rows/columns can be updated in parallel for some loops of the codes, and that it is also possible to update each row/column in parallel.

5.2 Barnes-Hut N-Body Simulation

This code is based on the algorithm presented in [1] which is used in astrophysics. In Fig. 10(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. Each leaf of the octree represents a subsquare which contains a single body and therefore points to this body stored in the $Lbodies$ list. Each octree node which is not a leaf has an array $child$ of eight pointers to its children.

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



(a) (b)

Fig. 10. Barnes-Hut data structure and compacted RSRSG.

All the traversals of the octree are carried out in the code by recursive calls. Due to the fact that our analyzer is still not able to perform an interprocedural analysis, we have manually carried out the inlining 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. 10 (a) and obtained in the corresponding RSRSG, Fig. 10 (b). The first step of the code, (i), is successfully analyzed in level L_1 but the best accurate description of the data structures used in steps (ii) and (iii) are obtained in level L_3.

However, regarding Table 1, there is paradoxical behavior that deserves explanation: L_3 expends less time and memory than L_1 and L_2. In L_3 *SHARED* and *SHSEL* remain false for more nodes and links which leads to more nodes and links being pruned during the abstract interpretation and graph compression phase of the symbolic execution of the statements. This leads to significantly reducing the number of nodes and graphs, which reduces memory and time requirements.

6 Conclusions and Future Work

In this work we have extended our shape analysis techniques to allow for the automatic detection of dynamic data structures based on arrays of pointers that we have called multiselectors. In order to accurately support multiselectors we propose the use of multireference classes and instances. On the one hand, the multireference classes point out which are the possible configurations of links that may coexist for a given statement. On the other hand, the instances are the key to focussing on the particular position of the array of pointers which is actually involved in a statement including a reference to a multiselector ($sel[i]$).

To validate these techniques we have implemented them in an analyzer which can be fed with C code and returns the data structures at each program point. This analyzer has reported very accurate descriptions of the data structures used in the tested codes, requiring a reasonable amount of memory and time. To the best of our knowledge there is no other implementation able to achieve such successful results for complex C codes like the ones presented here.

Information about data structure is critical in order to carry out further compiler optimizations such as locality exploiting or automatic parallelization. In the near future we will approach the issue of these additional compiler passes, but before this we want to tackle the recursive calls problem.

References

1. J. Barnes and P. Hut. *A Hierarchical O(n· log n) force calculation algorithm*. Nature v.324, December 1986.
2. 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.
3. Francisco Corbera. *Automatic Detection of Data Structures based on Pointers*. Ph.D. Dissertation, Dept. Computer Architecture, Univ. of Málaga, Spain, 2001.
4. F. Corbera, R. Asenjo and E.L. Zapata *Accurate Shape Analysis for Recursive Data Structures.* 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, 2000.
5. F. Corbera, R. Asenjo and E. Zapata *Progressive Shape Analysis for Real C Codes.*, IEEE Int'l. Conf. on Parallel Processing (ICPP'2001), pp. 373-380. Valencia, Spain, September 3-7, 2001.
6. R. Ghiya and L. Hendren. *Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C.* In Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 1-15, St. Petersburg, Florida, January, 1-24, 1996.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.