On the Parallelization of Irregular and Dynamic Programs [⋆]

Oscar Plata, Rafael Asenjo, Eladio Gutiérrez,
Francisco Corbera, M. Angeles Navarro and Emilio L. Zapata

*Department of Computer Architecture, University of Málaga,
29071 Málaga, SPAIN*

Abstract

Current compilers show ineffective when optimizing complex applications, both analyzing dependences and exploiting data locality and extracting parallelism. Complex applications may be characterized as irregular and dynamic. Irregular applications arrange data as multi-dimensional arrays and memory is referenced through array indirections. Dynamic applications organize data as pointer-based structures (lists, trees, ...) and memory is referenced through pointers. In this paper we discuss a methodology we designed to develop efficient parallelization techniques for irregular and dynamic applications, that proceeds in three stages: Recognizing the complex program structure, data analysis and program parallelization based on code/data transformations. Two case examples are analyzed in detail in the context of this methodology: Irregular reductions and shape analysis for dynamic data structures.

Key words: Irregular programs, dynamic programs, pointer-based data structures, compilers
PACS: ??.??

1 Introduction

Current automatic parallelizers obtain reasonably efficient parallel codes from most of the regular applications. Such applications deal with data organized as multi-dimensional arrays, and most of the computations are arranged as uniform nested loops. However, the compiler efficiency is generally much lower for other kind of programs, those that include complex computation and/or

⋆ This work was supported by Ministry of Education and Culture (CICYT), Spain, through grant TIC2000-1658

```
do i = 1,N
   ...
   compute ξ
   ...
   A(f(i)) = A(f(i)) ⊕ ξ
   ...
enddo
```

(a)

```
while (condition)
{
   ...
   p→data = value;
   p = p→next;
   ...
}
```

(b)

Fig. 1. Example of an irregular computation (a) and a dynamic computation (b)

data structures. In the presence of such programming complexities compilers usually run into trouble both analyzing dependences and exploiting data locality and extracting parallelism.

We may distinguish two important classes of complex applications: irregular and dynamic. Irregular applications are characterized by the fact that data is structured as multi-dimensional arrays, as in regular applications, but it is referenced through array indirections (arrays with subscripted subscripts). These applications are typically coded using procedural languages like Fortran77. Dynamic applications, on the other hand, deal with data organized as complex, pointer-based structures (lists, trees, ...), and it is referenced through pointers. Typical applications of this class are coded using languages like C/C++, Java or Fortran90.

Figure 1 shows example codes for irregular and dynamic computations. The first piece of code represents an irregular histogram reduction, where a reduction array (A) is updated at some points given by the indirection array (f). A key issue in the parallelization of this loop includes solving the possible cross-iteration true data dependences due to the indirection array. For instance, if array f is not a permutation we will have such dependences. The second code corresponds a variable loop where a pointer-based data structure is updated. Now, in the parallelization of this loop we have to solve possible cross-iteration dependences due to cycles in the pointer-based list.

In this paper we discuss our recent work about developing efficient parallelization techniques for irregular and dynamic applications. Basically our techniques are enclosed into a broad parallelization method, that can be broken down into several phases: recognizing the irregular/dynamic structure of the code, data analysis, and selection of an ad-hoc parallelization technique fulfilling some performance properties.

We present some of our recent advances in this field. In particular, we designed a methodology to parallelize codes with irregular reductions exploiting data locality. From this methodology we derived a number of efficient locality oriented run-time parallelizing techniques. On the other hand, we developed new shape analysis techniques for pointer-based data structures to enable dependence analysis in dynamic codes. Such techniques may be used to analyze memory references needed to develop efficient optimization and parallelization methods for dynamic codes.

The rest of the paper is organized as follows. section 2 discusses the methodology we use to develop our optimization and parallelization compilation techniques for irregular/dynamic codes. Next, specific techniques for a widely found irregular computational structure, named irregular reduction, is described. Shape analysis techniques for dynamic data structures are analyzed in section 4. Finally, conclusions are drawn.

2 Parallelization methodology for irregular/dynamic codes

This section describes a methodology for the efficient exploitation of the available parallelism in programs with irregular and/or dynamic computation/data structures. We developed techniques to discover certain program (code and data) properties that are essential in the effective optimization, as well as parallelization methods that take advantage of such properties. The parallelization methodology proceeds in several stages, as follows:

(1) *Program structure:* Analysis of the computational structure of the program, as well as the data structures used. As a result of this analysis we can recognize the irregular and/or dynamic nature of the program.
(2) *Data analysis:* A complete data analysis is needed to determine whether parallelism is exploitable, or to enable some optimizations. It is also needed to know where and how such parallelization/optimization can be done. In case of irregular and dynamic programs, this stage becomes very complex. Two important tasks included into this stage are both the analysis of the data structure and the analysis of memory references. The first analysis determines how data is organized and the relationship among different data items. The second analysis discovers how data is referenced and the relationship among these data references.
(3) *Program parallelization:* Information resulting from program structure and data analysis allows to decide what specific parallelization method is best suited to be used. We are specially interested in the development of methods that optimize some important program properties, like data locality or communication overhead.

In the rest of the paper we describe two representative case studies in the context of the considered parallelization methodology. The first case study, that constitutes an important class of irregular programs, corresponds to codes with irregular reductions. For these codes the three stages in the parallelization

```
        REAL    A(1:ADim)
        INTEGER f_1(1:N_1, 1:N_2,... ,N_{nLoops})
        INTEGER f_2(1:N_1, 1:N_2,... ,N_{nLoops})
                ...
        INTEGER f_{nInd}(1:N_1, 1:N_2,... ,N_{nLoops})

    h:  do i_1 = 1,N_1
            do i_2 = 1,N_2
                ...
                do i_{nLoops} = 1,N_{nLoops}
                    Compute $\xi_1, \xi_2, \ldots \xi_{nInd}$
                    A(f_1($i_1, i_2, \ldots i_{nLoops}$)) = A(f_1($i_1, i_2, \ldots i_{nLoops}$)) + $\xi_1$
                    A(f_2($i_1, i_2, \ldots i_{nLoops}$)) = A(f_2($i_1, i_2, \ldots i_{nLoops}$)) + $\xi_2$
                    ...
                    A(f_{nInd}($i_1, i_2, \ldots i_{nLoops}$)) = A(f_{nInd}($i_1, i_2, \ldots i_{nLoops}$)) + $\xi_{nInd}$
                enddo
                ...
            enddo
        enddo
```

Fig. 2. Nested loop with multiple irregular reductions

methodology will be discussed. The second case study will focus on the second stage, data analysis, for general dynamic codes processing pointer-based data structures.

3 Programs with irregular reductions

Many common data organizations used in numerical applications involve irregular memory accesses, in which array elements are referenced by means of indirections. Reduction operations are often found in the context of irregular codes in scientific and numerical applications, representing an important class of irregular problems. Reduction operations are based in commutative and associative operators, like additions, multiplications, and so on.

An example of a piece of code carrying out multiple irregular reductions inside a nested loop is shown in Figure 2 (it is also known as histogram reduction). $A()$ represents the reduction array (that could be multidimensional), which is updated (the reduction operation is an addition in this example) by means of the subscript arrays $f_1()$, $f_2()$, ... Terms ξ_1, ξ_2, ... represent effective computation.

Considering the parallelization methodology described in the previous section, the first stage corresponds to the recognition of the irregular reduction and then what arrays work as reduction array(s) and which ones as subscript arrays. This stage may be accomplished in a compiler through the use of pattern-matching or idiom recognition techniques [17,2].

Once irregular reductions have been recognized, a data analysis of the code proceeds. As shown in Figure 2, all relevant data (from the viewpoint of this stage) is organized as arrays, so no further data structure analysis is needed. We next proceed to analyze memory references. Due to the subscripted subscripts, loop–carried data dependences may be present, and they cannot be detected at compile time (due to the subscript arrays). Techniques have been developed to detect this kind of data dependences at run-time [19].

However, because of the associative and commutative properties satisfied by the reduction operator, the possible data dependences due to the array reductions may be overcome by code/data transformations. Such transformations corresponds to the third stage in our methodology. In the last few years various code/data transformations that parallelize irregular reduction loops appeared in the literature (see related work). In the next sections we will discuss a framework to develop efficient parallelization techniques for irregular reduction loops. This framework is focused to exploit data locality on shared–memory multiprocessor platforms.

3.1 Locality and affinity

In order to optimize data locality through code/data transformations, we first need to characterize it. Without loss of generality, let us take the reduction loop shown in Figure 2 as a working example. We can distinguish two sources of data locality: Read locality associated with accesses to read-only and privatizable arrays, and write locality associated with accesses to the reduction arrays.

In (cache-coherent) shared memory multiprocessors, writes usually have a stronger impact on performance overhead than reads (writes must propagate and serialize through the memory hierarchy). So it is much important, from the performance viewpoint, to optimize writing locality.

We distinguish between two classes of write locality: Intra–iteration and inter–iteration. Intra–iteration locality corresponds to write locality inside the same nested loop iteration. Inter–iteration locality, on the other hand, is due to writes on the reduction arrays executed on different loop iterations.

When parallelizing the reduction code, the class of locality we can exploit depends on the granularity of the parallelization method. It is usual that the minimum amount of partitionable code is one full loop iteration. In such case, only inter–iteration locality can be exploited by code parallelization. If we want to also exploit intra–iteration locality, we must resort to data reorganizations [13] (basically the contents of the subscript arrays).

A simple method to exploit inter–iteration locality proceeds in two steps: First, we state a data distribution of the reduction arrays among all threads that cooperate in the parallel computation. Second, reduction loop iterations are assigned to threads in such a way that the number of local writes (writes to owned reduction array elements) is maximized. Note that these iteration assignments not only exploit locality but also avoid the need of run-time dependence analysis, as iterations from different threads can be executed with no write conflicts.

In what follows we will describe a framework to define efficient locality–based loop iteration assignments.

First, we need some definitions. Without loss of generality, let us consider the reduction loop in Figure 2. $A(1{:}ADim)$ represents the reduction array, which is updated inside a nested loop, being $\vec{\imath} = (i_1, i_2, ...i_{nLoops})$ the iteration index vector. Also let $P = \{1, 2, ...nThreads\}$ be the set of threads identifiers that cooperate in the computation, and let $\Psi : \{A(1), A(2), ...A(ADim)\} \rightarrow P$ be a distribution function of the array A on the threads.

Definition 3.1 *The **write access set** of the iteration $\vec{\imath}$ is defined as the set of indices m such that $A(m)$ is written in such iteration. The write access set is denoted as $Acc_{\vec{\imath}}(A)$, and thus $Acc_{\vec{\imath}}(A) = \{m \in [1, ADim] \mid A(m)$ is written in iteration $\vec{\imath}\}$.*

Definition 3.2 *Two iterations, $\vec{\imath}$ and $\vec{\jmath}$, are **write affine** if their write access sets are mapped to the same subset of threads, that is, $\Psi(Acc_{\vec{\imath}}(A)) = \Psi(Acc_{\vec{\jmath}}(A))$.*

Definition 3.3 *Two iterations, $\vec{\imath}$ and $\vec{\jmath}$, are **write dissimilar** if their write access sets are mapped to disjoint subsets of threads, that is, $\Psi(Acc_{\vec{\imath}}(A)) \cap \Psi(Acc_{\vec{\jmath}}(A)) = \emptyset$.*

3.2 Write affinity based parallelization

Using the write affinity property defined in the previous section we will derive an optimal method to parallelize histogram reduction loops. Given a data distribution function of the reduction array, a code transformation of the reduction loop will be defined so as some performance issues are optimized: parallelism and data locality are maximized, and computation replication, memory overhead, extra workload and synchronization overhead are minimized.

Let us start with new definitions. Definition 3.2 in previous section states a binary relation between two iterations, given a data distribution function of the reduction array. Such relation will be called ***affinity relation***. It is easy to see that the affinity relation is an equivalence relation, that is, it satisfies reflexive, symmetric and transitive laws. So, equivalence classes can be defined.

Definition 3.4 *Given the affinity relation, an equivalence class is a subset of write affine iterations, that is, iterations with their access sets mapped to the same subset of threads. Given Q a subset of P, let $\mathcal{C}_Q$ be an* ***affinity equivalence class****, then $\mathcal{C}_Q = \{\vec{\imath} \in \mathcal{S} \mid \Psi(Acc_{\vec{\imath}}(A)) = Q\}$, where $\mathcal{S}$ is the set of iterations.*

Definition 3.5 *The set of all affinity equivalence classes in the iteration set $\mathcal{S}$ is called the* ***affinity quotient set****, and denoted as $\mathcal{S}/\mathit{aff}$.*

When using some locality-oriented data distribution function Ψ, for example a classical block distribution, it would be possible to exploit write inter–iteration locality by considering those iterations belonging to a same affinity equivalence class. From the parallelization viewpoint, we need to distinguish data independent reduction iterations.

Definition 3.6 *Two affinity equivalence classes, $\mathcal{C}_Q$ and $\mathcal{C}_R$, are defined* ***dissimilar*** *if two iterations, $\vec{\imath} \in \mathcal{C}_Q$ and $\vec{\jmath} \in \mathcal{C}_R$, are write dissimilar.*

Lemma 1 *Two classes, $\mathcal{C}_Q$ and $\mathcal{C}_R$, are dissimilar if and only if $Q \cap R = \emptyset$.*

In a reduction loop the only true data dependences are caused by writes in the reduction array, thus two write dissimilar iterations are assured to be data independent. Hence iterations belonging to dissimilar equivalence classes can be executed fully in parallel, with no write conflicts. That means that it would not be any parallelization overheads, like extra memory, synchronizations or computation replication. These are precisely the issues that we want to minimize in the parallelization of the reduction loop. In addition, if we can find large sets of dissimilar equivalence classes, we would have a lot of exploitable parallelism.

Definition 3.7 *The* ***dissimilarity graph****, denoted as $DG(\mathcal{S}/\mathit{aff}) = (N_{DG}, E_{DG})$, is defined as an undirected graph whose vertices are affinity equivalence classes, that is, N_{DG} is the affinity quotient set $\mathcal{S}/\mathit{aff}$. There exists an edge between two classes in the graph if such classes are not dissimilar.*

The dissimilarity graph relates potentially data dependent reduction iterations, for a given data distribution function. Non directly connected vertices in that graph corresponds to dissimilar equivalence classes, that contain data independent iterations. Therefore, if we want to maximize exploitable parallelism, we have to find the maximum number of non directly connected vertices in the dissimilarity graph. This can be done by applying a vertex coloring al-

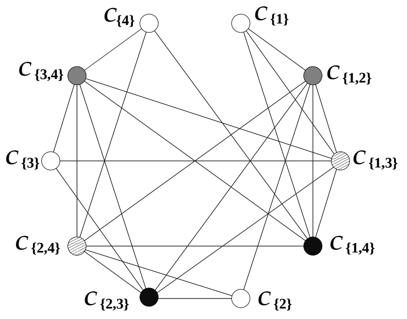


Fig. 3. Vertex coloring of a dissimilarity graph

```
for $color \in \mathcal{DG}(\mathcal{S}/\mathsf{aff})$
        forall $\mathcal{C} \in color$
                Execute iterations $\in \mathcal{C}$
        end
        C$barrier
end
```

Fig. 4. Parallel reduction loop based on affinity classes

gorithm to the graph. Resulting from the coloring process sets of dissimilar classes are obtained. Iterations from the classes in each one of these sets can be executed fully in parallel because they write in non conflicting areas of the reduction array.

As an example, consider a reduction loop with two reductions (indirections), one reduction array, four threads, and a certain distribution function Ψ. In this case, the maximum possible number of equivalence classes in the affinity quotient set is 10, and would be as follows:

$$\mathcal{S}/\text{aff} = \{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, \mathcal{C}_{\{3\}}, \mathcal{C}_{\{4\}}, \mathcal{C}_{\{1,2\}}, \mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{1,4\}}, \mathcal{C}_{\{2,3\}}, \mathcal{C}_{\{2,4\}}, \mathcal{C}_{\{3,4\}}\}.$$

The resulting dissimilarity graph is shown in Figure 3. After applying the vertex-coloring algorithm to this graph we can obtain the sets of classes than can be executed concurrently. Vertices with the same color in the graph are not directly connected. Therefore, sets of dissimilar equivalence classes can be obtained by grouping together all classes with the same color, that is:

$$\Big\{ \{\mathcal{C}_{\{1\}}, \mathcal{C}_{\{2\}}, \mathcal{C}_{\{4\}}, \mathcal{C}_{\{4\}}\}, \{\mathcal{C}_{\{1,2\}}, \mathcal{C}_{\{3,4\}}\}, \{\mathcal{C}_{\{1,3\}}, \mathcal{C}_{\{2,4\}}\}, \{\mathcal{C}_{\{1,4\}}, \mathcal{C}_{\{3,2\}}\} \Big\}.$$

We can schedule a parallel execution of the reduction loop following an inspector/executor scheme. An inspector builds the affinity equivalence classes, the corresponding dissimilarity graph and color it. After the inspection stage, computations are scheduled by the executor as shown in Figure 4. Iterations in equivalence classes with the same color are executed in parallel, while a synchronization point is placed between execution of sets of classes with different colors.

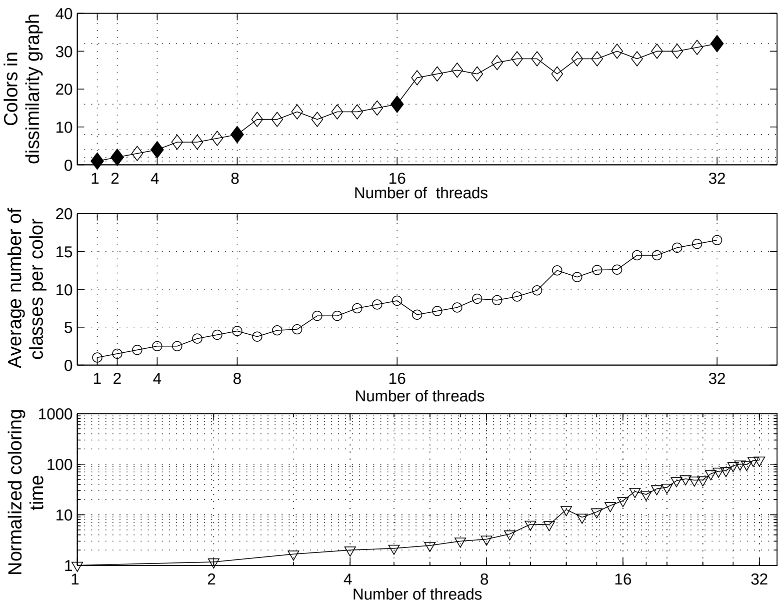


Fig. 5. Computation of dissimilarity graph coloring for a loop with two indirections

3.3 Compiler implementation

Although the general and theoretical approach described in the previous section could be used in parallelizing reduction loops, however some serious difficulties arise in practice. To maximize the available parallelism the minimum number of colors in the dissimilarity graph has to be found. This minimum number of colors is called the vertex-chromatic index of the graph, and it is known that a general algorithm to compute it is NP-hard. Nevertheless some simplifications can provide a non-optimal coloring with a polynomial complexity. In addition, to reduce the number of colors certain restrictions would be desirable, like maximizing the size of the equivalence classes with the same color, or considering conditions for workload balance. Such operations, however, would increase significantly the overhead of the inspection stage. An addition difficulty is the fact that the number of possible non-empty affinity classes grows rapidly with the number of indirections in the reduction loop.

For example in Figure 5 it is shown the results for the dissimilarity graph color computation when two indirections are considered. A greedy coloring algorithm [9] has been applied, using different initial vertex orders. An important fact is that a optimum number of colors is obtained if the number of threads is a power of two. For these cases the number of colors is equal to the number of threads. As it is seen in Figure 5 the coloring time follows a complexity $\mathcal{O}(nThreads^4)$, being $nThreads$ the number of threads.

In order to make practical the implementation of the method in a compiler, the inspection phase must be lightened. This fact can be achieved by simplifying

		2 indi-rections	3 indi-rections	4 indi-rections
4 threads	Privatization	4.5	1.9	2.4
	Local–Write	10.6	6.4	1.9
	DWA–LIP	11.5	7.2	5.0
8 threads	Privatization	7.8	2.0	2.5
	Local–Write	11.3	11.8	4.9
	DWA–LIP	12.5	12.6	7.7

Table 1
Speedups for the EULER code using privatization, Local–Write and DWA–LIP

the equivalence class building process.

We have developed an approach called *Data Write Affinity with Loop Index Prefetching* (DWA–LIP) that is based both on a block data distribution function and on a restricted definition of the affinity relation. Instead of using a generic subset of threads, Q, to characterize an affinity equivalence class, $\mathcal{C}_Q$, DWA–LIP uses a pair of parameters $(B_{min}, \Delta B)$, being $B_{min} = min(Q)$ and $\Delta B = max(Q) - min(Q)$. The dissimilarity test with the new affinity relation is simpler since two iterations will be write dissimilar when their pairs $(B_{min}, \Delta B)$ do not correspond to overlapped areas of the reduction array.

This simplification in the definition of the affinity relation has a negative effect because there are pairs of write dissimilar iterations that no longer are recognize as such with the new definition. This reduces the detected parallelism to be exploited in the execution phase. Nevertheless the new simplified affinity relation allows the inspector to be lighter and makes possible an efficient schedule of the execution of dissimilar classes during the execution phase [10,11].

In Figure 1 experimental results for different methods are shown (see related work). A code that implements Euler differential equations has been used. It contains some reduction loops with 2, 3 and 4 indirections, carrying out magnitude computations over edges, faces and tetrahedra, respectively. The input data correspond to a mesh description of 800Knodes and connectivity 18. We have tested a privatization-based method, in particular Array Expansion [15], and two affinity-based methods, Local–Write and DWA–LIP. Since input data exhibit a low inter–iteration locality the privatization-based method has a low efficiency in contrast to affinity-based methods. When the number of indirections grows the intra–iteration locality is poorer and thus the performance of all methods decreases.

3.4 Related work

One of the most popular methods to parallelize reduction loops is based on the privatization of the reduction arrays [1]. This way, iterations become data independent (no write conflicts) allowing a free scheduling of iterations in the threads. Although several versions and optimizations of these methods were proposed [15], privatization-based techniques have important drawbacks, like a large extra memory requirement (reduction arrays must be replicated on all threads) and no exploitation of data locality.

Instead of distributing loop iterations, another group of techniques uses distribution of the reductions arrays. This approach avoids the extra memory overhead discussed previously, and make possible to take data locality into consideration. In these methods iterations are partitioned and assigned to the threads on the basis of a previously chosen data distribution for the reduction arrays. However, some specific technique must be used to solve data dependences due to write conflicts in the reduction arrays [12,10,11].

The approach called Local–Write [12] parallelize reduction loops exploiting write locality, as with DWA–LIP. However, this method is based on applying loop-splitting to those iterations belonging to affinity classes $\mathcal{C}_Q$ with $Card(Q) > 1$ (that is, Q has two or more threads). For these split iterations the computations are replicated which implies an effective loss of parallelism.

4 Analysis of Dynamic Programs

4.1 Motivation of the shape analysis

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. This is due to 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 pointer-based codes.

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. Therefore, this work is part of the first step (program structure analysis) of our parallelization methodology.

There are other open research lines dealing with the analysis of codes in the presence of pointers, such as alias analysis or points-to analysis. Basically, these analysis are designed to determine the superset of locations to which a pointer *must* or *may* point (points-to sets) [7]. These kinds of pointer analysis provide enough information to allow for some scalar optimizations, such as Common Subexpression Elimination, Loop Invariant Removal, or Location Invariant Removal [8]. However, the information provided by the points-to sets is not accurate enough to enable more ambitious optimizations such as loop-level automatic parallelization, automatic data distribution, and locality exploiting. Currently, the majority of research groups rely on manual annotations when dealing with such complex code optimizations in the presence of pointers, due to points-to analysis is not sufficient. For instance, Chilimbi et al. ask the programmer to annotate the code to exploit cache locality [4] or a previous execution profile is needed in order to exploit cache prefetching [3]. In the area of distributed memory locality exploiting and communication optimization, Zhu and Hendren [22] also rely on code annotations with special compiler directives. Similarly, Rogers et al. [18] propose a thread-level parallelism in codes annotated with directives such as *futurecall* and *touch*.

However, some groups are trying to automatically extract more information from the code text to optimize codes based on pointers. For example, Ghiya [8] have implemented the McCAT compiler to put pointer analysis to work. Basically, this compiler uses points-to analysis to deal with stack-directed pointers and connection analysis and shape analysis to deal with heap-directed pointers. This analysis is used for exploiting two parallelism levels in codes based on recursive data structures which do not change their shape while they are traversed: at the function level when routines traverse disjoint sub-tree structures; and at the loop level in two cases; single liked list traversing and array of pointers to disjoint structures traversing. However, their shape analysis is too simple and conservative leading to a serious lack of parallelism exploitation. This is mainly due to it does not keep information about the topological structure of the links between heap locations.

Thus, we have to emphasize that our final goal is to allow for the automatic optimization of codes based on recursive data structures, but it is clear that, first of all, better shape analysis techniques have to be proposed. That is, new approaches to automatically capture the essential characteristics and properties of heap-allocated data structures are essential. With this in mind, our proposal is based on approximating all the possible memory configurations that can arise after the execution of a statement by a set of graphs: the *Reduced Set of Reference Shape Graphs* (RSRSG). With our framework we can achieve accurate results in a reasonable analysis time and expending a reasonable ammount of memory. Besides this, we cover situations that were previously unsolved, such as detection of complex structures (arrays of pointers, lists of trees, lists of lists, etc.) and structure permutation, as we will see in the next sections.

4.2 Method overview

Basically, our method is based on approximating by graphs 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*.

Note that due to the control flow of the program, a statement could be reached by following several paths in the control flow. Each "control path" has an associated memory configuration which is modified by each statement in the path. Therefore, a single statement in the code modifies all the memory configurations associated with all the control paths reaching this statement. Each memory configuration is approximated by a graph we call *Reference Shape Graph* (RSG). So, taking all this into account, we conclude that each statement in the code will have a set of RSGs associated with it.

4.2.1 RSGs and node properties

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. 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. These properties are briefly described here, but a more detailed description can be found in [6]:

1. Type: This property states the data type of the memory locations represented by a node.

2. Structure: This information avoids the summarization into the same node of memory locations belonging to non-connected data structures (i.e. both data structures do not share any element).

3. Simple Paths (SPATH): This property avoids the summarization of memory locations near pointer variables. Since data structures are accessed and modify via pointer variables, by keeping a precise description of the memory location near the pointer variables the compiler will carry out a more accurate shape analysis.

4. Reference Patterns: For each node, this property is represented by two sets: *SELINset* contains the selectors which reference the node from other nodes and *SELOUTset* contains the selectors which point from this node to others. For example, in a doubly linked list, a node representing the last item of the list has $SELINset=\{next\}$ and $SELOUTset=\{prev\}$, because *next* is an "input" selector reaching the node and *prev* is an "output" selector leaving the node. Only nodes with similar reference patterns can be summarized into a single one.

5. 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. 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.

6. Touch Information: This property is taken into account only inside loop bodies to avoid the summarization of already visited locations with non-visited ones.

7. Cycle Links: This information is introduced 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 node n is reached again.

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 this set

Reduced Set of Reference Shape Graphs (RSRSG), since not all the different RSGs arising in each statement 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 statement are limited too. This union of RSGs greatly reduces the number of RSGs and leads to a practicable analysis.

4.2.2 Generating the RSRSGs: the symbolic execution

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.

Let us illustrate all this with an example. In Figure 6 we can see a simple code with seven pointer statements. Our analyzer symbolically executes each statement to build the RSRSG associated with them. Actually, after the execution of the third statement we obtain an RSRSG with a single RSG which represents three different memory locations by three nodes; all of them of the same type, with the same *nxt* selector, but pointed to by different pointer variables *(pvars)*. Now, this RSRSG is modified in three different ways because there are three different paths in the control flow graph, each one with a different pointer statement. All these paths join in statement 7, and after the execution of this statement we obtain an RSRSG with two RSGs. This is because the RSGs coming from statements 5 and 6 are compatible and can be summarized into a single one.

The whole symbolic execution process can be seen by looking at Fig. 7. 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

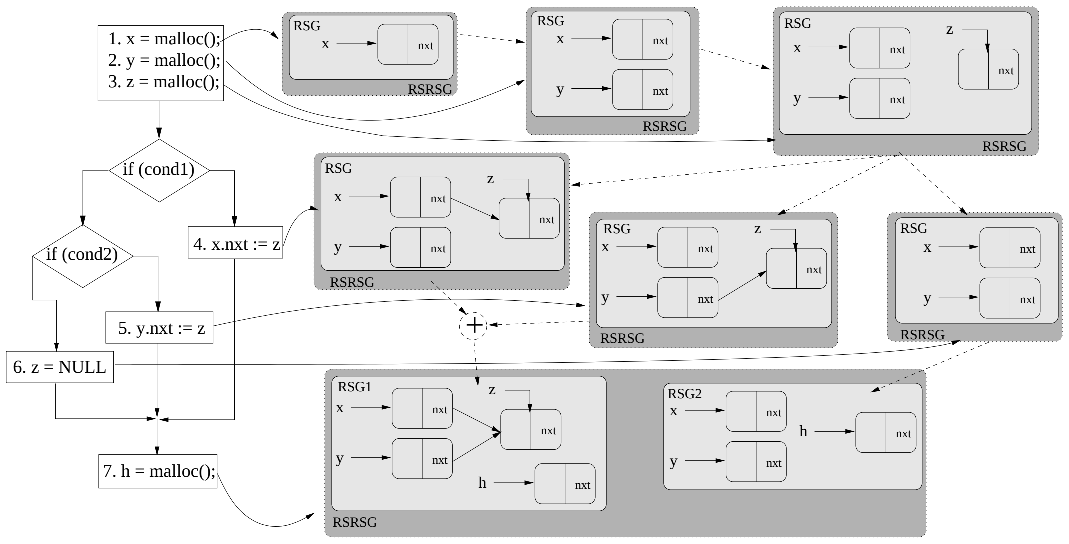


Fig. 6. Building an RSRSG for each statement of an example code

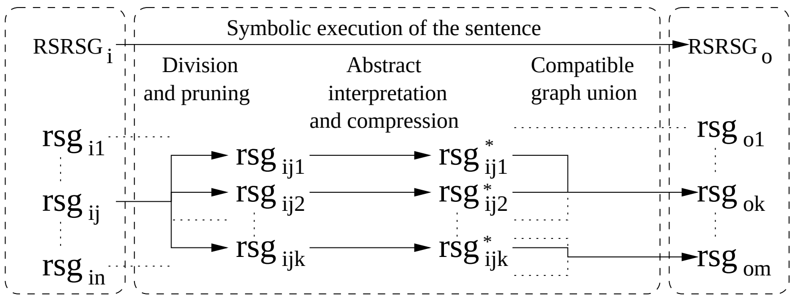


Fig. 7. Schematic description of the symbolic execution of a statement.

greatly reduces the number of RSGs in the resulting RSRSG.

The abstract interpretation is carried out iteratively for each statement until we reach a fixed point in which the resulting set of rsg_{oj} associated with the statement does not change any more. This way, for each statement that modifies dynamic structures, we have to define the abstract semantics which describe how these statements modify the rsg_{ij}. 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 formally describe the abstract semantics of each one of these statements. However, we intuitively present the modifications involved in an RSG after the statement symbolic execution:

- The $x = NULL$ statement 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 $x = malloc$ statement 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 statement and the previous one, we always automatically insert the $x = NULL$ statement to ensure that before assigning a new value to x, x is not pointing to any place.
- The statement $x \rightarrow sel = NULL$ 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 by 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.

  All these processes can be better illustrated by a simple example. In Fig. 8 (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 $x \rightarrow nxt = NULL$ statement where x is pointing to the memory location represented by node n_1. As we said, the first step in the abstract interpretation of this statement is the division operation. Figure 8 (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 $x \rightarrow nxt$. In Fig. 8 (c) we show the result of the pruning process in which the compiler removes nodes and links which do not fulfill the graphs' properties. In fact, $rsg\prime\prime_1$ represents a list of three or more elements and $rsg\prime\prime_2$ is clearly a list of just two items. 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. 8 (d). Finally, we see in Fig. 8 (e) how we safely remove the link $x \rightarrow nxt$ in both graphs to obtain the final rsg_1 and rsg_2.
- The $x \rightarrow sel = y$ statement implies the execution of the same procedure just described for the $x \rightarrow sel = NULL$ statement (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 statement $x = y \rightarrow sel$ 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.

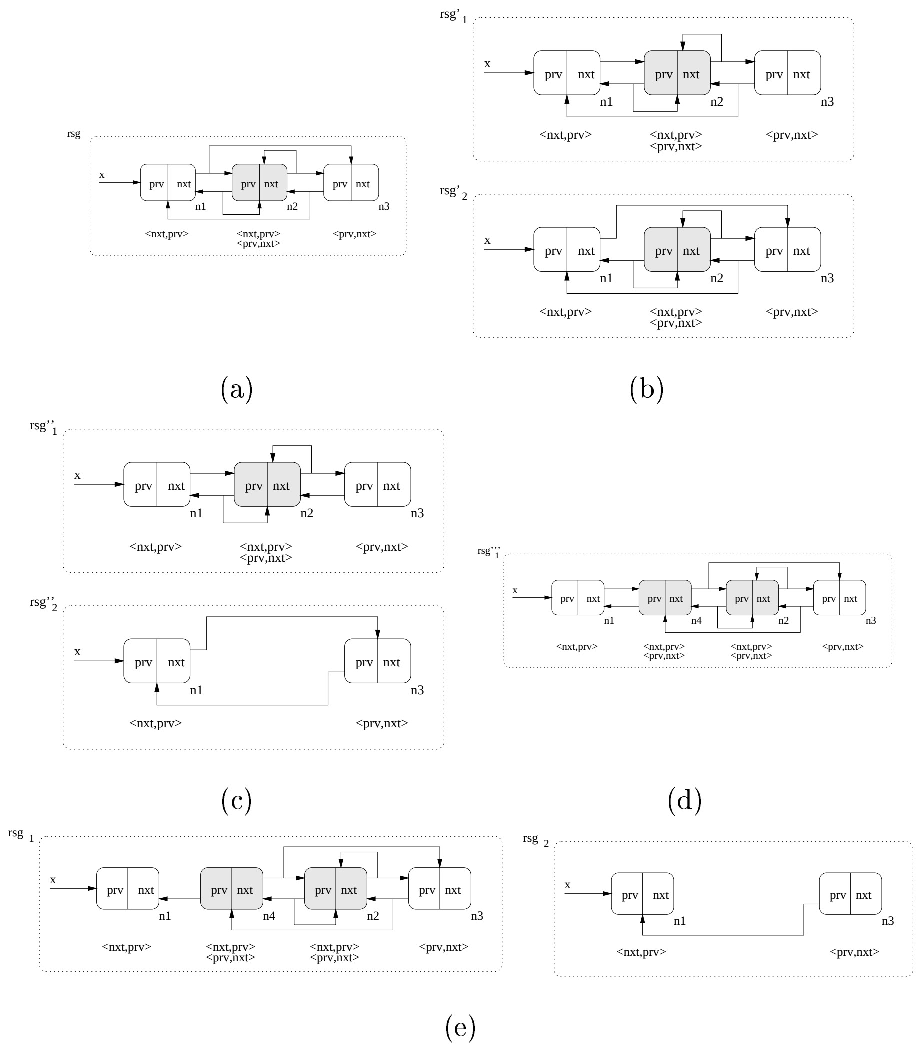


Fig. 8. Complete process of the abstract interpretation required by the $x \rightarrow nxt = NULL$ statement.

4.2.3 Dealing with arrays of pointers: 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 before, 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 9 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

```
typedef struct str {
    ...
    struct str1 *sel;
    struct str2 *sel1[256];
    struct str **sel2;
}
x=(str *)malloc(sizeof(struct str));
x->sel2=(str **)malloc(n*sizeof(str
*));
```

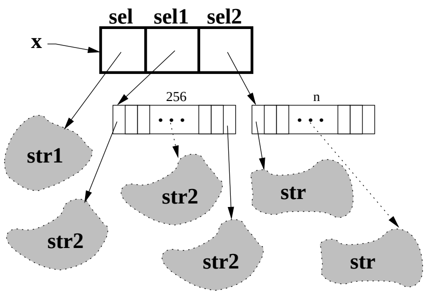


Fig. 9. Example of data structure containing arrays of pointers

analyzer. However, *sel*1 and *sel*2 represent arrays of selectors. The difference between *sel*1 and *sel*2 is that we know the size of the *sel*1 array at compile time, but the size of *sel*2 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 *sel*1 and *sel*2 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 the following procedure: 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 *sel*1), but before applying the symbolic execution, our analizer start by identifying the particular sel1[i] which is going to be updated, to subsequently proceed with the abstract interpretation.

4.3 Experimental results

Our RSRSG analyzer has been written in C and can be fed with an input code to generate the RSRSG associated with each statement of the code. The codes have to be preprocessed in a first step to just keep the statements dealing with pointers. 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 analysis comprises three levels: L_1, L_2, and L_3, from less to more complexity as we explain in [6].

With this tool we have analyzed several codes: an artificial code that we call "working example", the sparse Matrix by vector multiplication, the sparse Matrix by Matrix multiplication, the Sparse LU factorization, and the Barnes-Hut code. These five codes have two implementations, one in which arrays of pointers are implemented by doubly linked lists and the other in which arrays

	Working Ex.	S.Mat-Vec	S.Mat-Mat	S.LU	Barnes-Hut
Level	L_1 / L_2 / L_3	L_1 / L_2 / L_3	L_1 / L_2 / L_3	L_1 / L_2 / L_3	L_1 / L_2 / L_3
	Codes without arrays of pointers				
Time	0'03"/0'05"/0'06"	0'01"/0'02"/0'03"	0'20"/0'38"/1'00"	7'50"/-/-	5'56"/0'34"/2'06"
MBytes	2.11/2.78/3.02	1.37/1.85/2.17	8.13/11.45/12.68	99.46/-/-	37.82/8.82/8.94
Lines	213	104	156	164	216
	Codes including arrays of pointers				
Time	0'05"/0'07"/0'08"	0'01"/0'01"/0'01"	0'04"/0'06"/0'06"	1'08"/1'12"/-	23'08/25'27"/0'21"
MBytes	1.77/2.29/2.50	0.92/1.03/1.2	1.19/1.31/1.49	3.96/4.18/-	40.14/42.86/3.06
Lines	144	87	103	143	177

Table 2
Time and space required to process several codes with different number of code lines

of pointers are keep.

The first four codes were successfully analyzed in the first level of the analyzer, L_1. However, for the Barnes-Hut program the highest accuracy of the RSRSGs was obtained in the last level, L_3. All these codes where processed by our analyzer in a Pentium 4 1.6 GHz with 128 MB main memory. The time and memory required by the analyzer are summarized in Table 2. In this table we also show the number of code lines after the preprocessing of the original C codes. The particular aspects of these codes are described next.

(1) **Working example's RSRSG.** This code generates, traverses, and modifies the data structure presented in Figure 10 (a). A compact representation of the resulting RSRSG for the last statement of the code can be seen in Figure 10 (b). The data structure is a doubly linked list of pointers to trees (header list). Besides this, the leaves of the trees have pointers to 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.

    This data structure is built by a C code that also traverses the elements of the header list with two pointers and eventually can permute two trees. From the properties associated with the nodes in the RSRSG represented in Figure 10 (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. More precisely: (i) The analyzer successfully detects the doubly linked list which is acyclic by selectors nxt or prv and whose elements point to binary trees; (ii) Two different items of the header list cannot point to the same tree; (iii) 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.

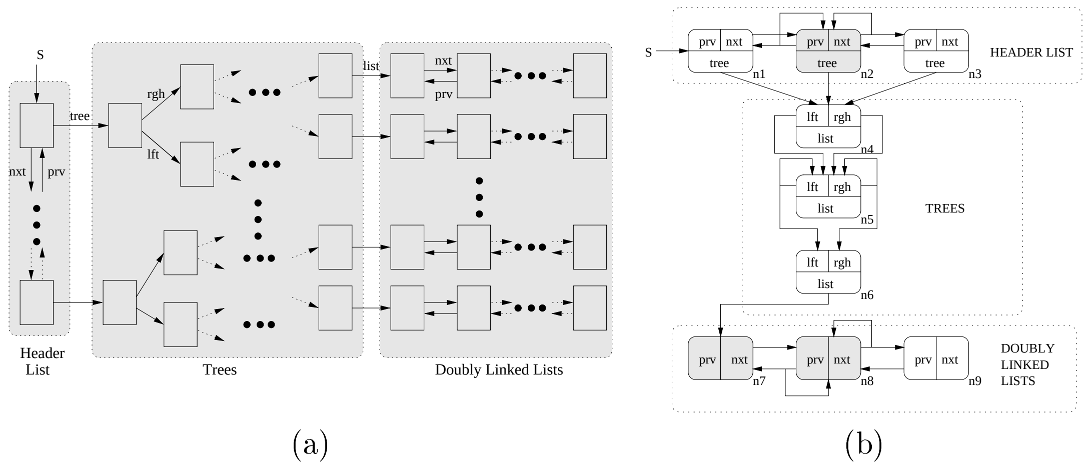


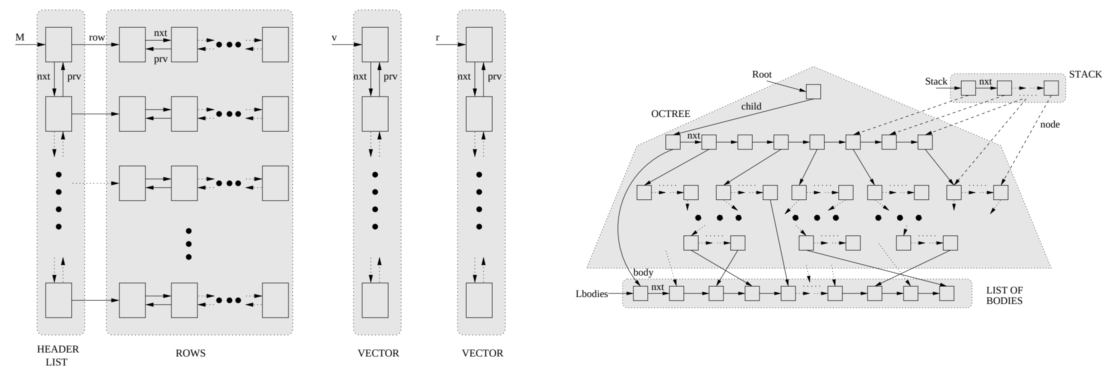
Fig. 10. A complex data structure (a), and compact representation of the resulting RSRSG (b)

The other implementation of this code is based on an array of pointers to the trees instead of the header list. Again, the analyzer can extract the same conclusions commented in the previous paragraph.

(2) **Sparse matrix codes.** Here we deal with some irregular codes which implements sparse matrix operations: the sparse matrix by vector multiplication, $r = M \times v$; the sparse matrix-matrix multiplication, $A = B \times C$; and the sparse LU factorization, $A = LU$.

The sparse matrices are stored in memory as a header doubly linked list (or an array of pointers) with pointers to other doubly linked lists which store the matrix rows (if the matrix is row-wise) or columns (for column-wise matrices). In figure 11 (a) we show the sparse matrix data structure for a row-wise matrix where the matrix header is implemented by an array of pointers. The sparse vectors, v and r are doubly linked lists. After the analysis process, carried out by our analyzer, the resulting RSRSG accurately represents the data structures. In the resulting RSRSG for the last statement of these codes we can identify the main properties of the data structures: (i) The rows of the matrix are pointed to from different elements of the header list/array; (ii) The doubly linked lists which store the rows of the matrices and the vectors are acyclic by selectors *nxt* and *prv*. A subsequent analysis of the code and the RSRSG associated with each statement would be able to state that several sparse matrix row can be traversed and updated in parallel and, in addition, it is also possible to update each row in parallel.

(3) **Barnes-Hut N-body simulation.** The structure used in this code is basically an octree where each leaf points to an element of a single linked list. In the implementation which avoids pointer arrays, each octree node which is not a leaf has a pointer *child* pointing to the first of its eight children which are linked by selector *next*. If pointer arrays are allowed, the pointers to the eight children are stored in an array of pointers, as we can



(a) (b)

Fig. 11. Data structure to store sparse matrices and vectors (a), and Barnes Hut main data structure (b)

see in figure 11 (b). The analysis of this code enable the parallel traversal of the octree which is precisely captured by the obtained RSRSG's.

4.4 Related work

There are several ways the shape analysis problem can be approached. We have focus in the graph-based methods in which the "storage chunks" are represented by nodes, and edges are used to represent references between them. For example, Plevyak et al. [16] have proposed the the "Abstract Storage Graph" (ASG), while Sagiv et al. [20] improved the ASG method with what they call "Static Shape Graphs" (SSG). In a previous work [5] we saw that ASG or SSG were not sufficient to deal with the complex data structures we presented in the previous section. Basically, ASG and SSG approaches were too imprecise and too conservative in many simple cases, due to they associate just one graph with each statement in the code. Besides, too much information is fused in a single node and then it is impossible to capture the real properties of the data structures represented by the graphs. We have overcome this drawback by considering several graphs per statement, while fulfilling some rules to avoid an explosion in the number of graphs and nodes in each graph.

A more recent work that also allows several graphs per statement is the one presented by Sagiv et al. [21]. 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. However, 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. There are several differences between our shape analysis method and that of Sagiv et al. [21]. The main one is that we join similar RSGs to build a reduced set of RSGs for each program point, while

		RSRSG		TVLA (sec.)	
Code	Lines	Time	Memory	Single	Multiple
create	9	Included in all		0.51	0.40
dellall	7	0.07s	133KB	0.42	0.44
delete	14	0.25s	179KB	2.73	5.18
fumble	10	0.09s	146KB	1.40	1.47
getlast+rot	11	0.14s	163KB	.78+.62	1.47+.88
insert	17	0.16s	197KB	2.86	2.77
merge	26	1.15s	387KB	8.25	12.01
reverse	10	0.15s	159KB	1.21	1.46
swap	8	0.09s	152KB	0.7	0.61
bublesort	32	2.52s	389KB	186.60	out of sp.

Table 3
Comparing RSRSG with TVLA

in [21] they keep all the graphs (multiple structure approach) or just one (single structure approach). We think that this may explain why their Three-Valued-Logic Analyzer (TVLA) runs out of memory for simple codes such as the singly linked list bubble sort using the multiple structure approach [14]. Besides, they recognize that their TVLA engine is only useful to analyze small programs and report experimental results for small, singly linked list operations (insert, reverse, sort, etc.), as we can see in Table 3. However, they have not published experimental results successfully dealing with real codes based on the combination of complex data structures such as doubly linked lists pointing to trees or to other lists, etc. In this Table 3 we also compare their Java-written TVLA running on a Pentium II-400MHz with our C-written analyzer on a Pentium III-500MHz.

5 Conclusions

This paper addresses the problem of automatic parallelization of irregular and dynamic applications. From our work on this problem we may derive two main conclusions. First, a complete and powerful data analysis is fundamental. This analysis must include, at least, two important tasks: Analysis of the data organization, and analysis of the memory references. In irregular codes, data organization analysis is not difficult as typically data is arranged as arrays. However, memory references are dynamic and data dependant. In dynamic codes, however, both analysis are very complex. In this line, we have developed shape analysis techniques to capture properties of complex pointer-based data structures.

The second conclusion is that we consider a promising way to obtain an effective parallelization to design ad-hoc techniques for specific complex computational structures. For instance, we discussed an efficient solution for irregular reductions. In a similar way, once the data organization of a pointer-based code has been identified, it is possible to develop efficient automatic techniques to traverse and update these data structures (trees, linked-lists, ...) in parallel.

References

[1] W. Blume, R. Doallo, R. Eigenmann, *et al.* Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[2] W. Blume and R. Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. *ACM Int'l Conf. on Supercomputing (ICS'94)*, pp. 528–537, 1994.

[3] T.M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. *ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'01)*, pp. 191–202, 2001.

[4] T.M. Chilimbi, M.D. Hill and J.R. Larus. Cache-conscious structure layout. *ACM SIGPLAN Conf. on Programming Languages Design and Implementation (PLDI'99)*, pp. 1–12, 1999.

[5] F. Corbera, R. Asenjo, and E.L. Zapata. New shape analysis for automatic parallelization of C codes. *ACM Int'l Conf. on Supercomputing (ICS'99)*, pp. 220–227, 1999.

[6] F. Corbera, R. Asenjo, and E.L. Zapata. Accurate shape analysis for recursive data structures. *Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'2000)*, pp. 1–15, 2000.

[7] M. Das. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices*, 35(5):35–46, 2000.

[8] R. Ghiya. Putting Pointer Analysis to Work. *PhD thesis*, School of Comp. Sci., McGill Univ., Montreal, 1998.

[9] A. Gibbons Algorithmic Graph Theory. Cambridge University Press, 1999.

[10] E. Gutiérrez, O. Plata and E.L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. *ACM Int'l. Conf. on Supercomputing (ICS'2000)*, pp. 78–87, 2000.

[11] E. Gutiérrez, O. Plata and E.L. Zapata. Improving Parallel Irregular Reductions Using Partial Array Expansion. *IEEE/ACM Int'l. Conf. for High Performance Computing and Communications (SC'2001)*, 2001.

[12] H. Han and C-W. Tseng. Efficient compiler and run-time support for parallel irregular reductions. *Parallel Computing*, 2000. 26(13–14):1861–1887.

[13] H. Han and C-W. Tseng. Improving locality for adaptive irregular scientif codes. *Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'2000)*, pp. 173-188, 2000.

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

[15] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular fortran programs. *4th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers (LCR'98)*, 1998.

[16] J. Plevyak, A. Chien and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. *Int'l. Workshop on Languages and Compilers for Parallel Computing (LCPC'93)*, pp. 37–57, 1993.

[17] W.M. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. *ACM Int'l Conf. on Supercomputing (ICS'95)*, pp. 444–448, 1995.

[18] A. Rogers, M.C. Carlisle, J.H. Reppy and L.J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 17(2):233–263, 1995.

[19] L. Rauchwerger and D. Padua. The privatizing DOALL test: A run-time technique for DOALL loop identification and array provatization. *ACM Int'l Conf. on Supercomputing (ICS'94)*, pp. 33–43, 1994.

[20] M. Sagiv, T. Reps and R. Wilhelm. Solving shape-analysis problems in laguages with destructive updating. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 20(1):1–50, 1998.

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

[22] Y. Zhu and L. Hendren. Locality analysis for parallel C programs. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 10(2):99–114, 1999.