Detecting Loop-Carried Dependences in Programs with Dynamic Data Structures

Angeles Navarro [*], Francisco Corbera, Adrian Tineo, Rafael Asenjo and Emilio L. Zapata

Dept. of Computer Architecture, University of Malaga, Campus de Teatinos, E-29071. Malaga, Spain.

Abstract

The problem of data dependences detection in codes based on dynamic data structures, is crucial to various compiler optimizations. The approach presented in this paper focus on detecting data dependences induced by heap-directed pointers on loops that access dynamic data structures. Knowledge about the shape of the data structure accessible from a heap-directed pointer, provides critical information for disambiguating heap accesses originating from it. The new approach is based on a previously developed shape analysis that maintains topological information of the connections among the different nodes (memory locations) in the data structure. As a novelty, our approach carries out abstract interpretation of the statements being analyzed, annotating memory locations with read/write information. This information will be later used in a very accurate data dependence test which we describe in this paper. We also discuss its application to several different benchmarks.

Key words: Dynamic data structures, Compiler optimizations, Data dependence test, Shape analysis, Abstract interpretation

[*] Corresponding author. Phone: +34 952132791. Fax: +34 952132790.
Email addresses: `angeles@ac.uma.es` (Angeles Navarro), `corbera@ac.uma.es` (Francisco Corbera), `tineo@ac.uma.es` (Adrian Tineo), `asenjo@ac.uma.es` (Rafael Asenjo), `ezapata@ac.uma.es` (Emilio L. Zapata).

1 Introduction

Optimizing and parallelizing compilers rely upon accurate static disambiguation of memory references, i.e. determining at compile time if two given memory references always access disjoint memory locations. This problem, known as data dependences detection is crucial to various compiler optimizations such as instruction scheduling, data-cache optimizations, loop transformations, automatic vectorization and parallelization. Unfortunately the presence of alias in pointer-based codes makes memory disambiguation a non-trivial issue. An alias arises in a program when there are two or more distinct ways to refer to the same memory location. Program constructs that introduce aliases are arrays, pointers and pointer-based dynamic data structures.

Over the past twenty years powerful data dependence analysis have been developed to resolve the problem of array aliases. The problem of calculating pointer-induced aliases, called *pointer analysis*, has also received significant attention over the past few years [17], [14], [3]. Pointer analysis can be divided into two distinct subproblems: stack-directed analysis and heap-directed analysis. We focus our research in the latter, which deals with objects dynamically allocated in the heap. An important body of work has been conducted lately on this kind of analysis. A promising approach to deal with dynamically allocated structures consists in explicitly abstracting the dynamic store in the form of a bounded graph. In other words, the heap is represented as a storage shape graph and the analysis tries to capture some shape properties of the heap data structures. This type of analysis is called *shape analysis* and in this context, our research group has developed a powerful shape analysis framework [2]. Shape analysis algorithms have demonstrated high precision over schemes that name objects based on location sites (as in [6], [10]), but have not been sufficiently exploited and put to work in optimization algorithms, such as the data dependence analysis which we address in the paper.

The approach presented in this paper focuses on detecting data dependences induced by heap-directed pointers on loops that access pointer-based dynamic data structures. Particularly, we are interested in the detection of the loop-carried dependences (henceforth referred as LCDs) that may arise between the statements in two iterations of the loop. Knowledge about the shape of the data structure accessible from heap-directed pointers, provides critical information for disambiguating heap accesses originating from them, in different iterations of a loop, and hence to provide that there are not data dependences between iterations.

Until now, the majority of LCDs detection techniques based on shape analysis [4], [9], use as shape information a coarse characterization of the data structure being traversed (Tree, DAG, Cycle). One advantage of this type of analysis is that it enables faster data flow merge operations and reduces the storage requirements for the analysis. However, it also causes a loss of accuracy in the detection of the data dependences, specially when the data structure being visited is not a "clean" tree, contain cycles or is modified along the traverse.

Our approach, on the contrary, is based on a shape analysis that maintains topological information of the connections among the different nodes (memory locations) in the data structure. In fact, our representation of the data structure provides us a more accurate description of the memory locations reached when a statement is executed. Moreover, as we will see in the next sections, our shape analysis is based on the abstract interpretation of the program statements over the graphs that represent the data structure at each program point. In other words, the new approach does not relies on a generic characterization of the data structure shape in order to prove the presence of data dependences. The novelty is that our approach symbolically interprets the statements of the loop being analyzed, and let us annotate the real memory locations reached by each statement with read/write information. This information will be later used in order to find LCDs in a very accurate data dependence test which we describe in this paper.

We also discuss the behavior and effectiveness of our test when applied to some sample programs. For these experiments we considered small custom-made programs and benchmark programs. In the first category, we created sample codes that traverse DAG and Cyclic data structures. In the second category, we considered two programs that calculate the product of disperse matrices and vectors. These structures are usually pointer-based to avoid storing the null elements. In addition, we studied `mst` and `em3d` from the Olden suite [1] and `twolf` from the SPEC CPU2000 suite [15]. In the light of these experiments, we believe that the new approach provides more accurate results when compared to previous techniques in the context of real applications.

Summarizing, the goal of this paper is to present the compilation algorithms which are able to detect LCDs in loops that operate with general pointer-based dynamic data structures, using as a key tool a powerful shape analysis framework. For it, we organize the paper as follows: Section 2 briefly describes the key ideas under our shape analysis framework. With this background, in Section 3 we present the compiler techniques to automatically identify LCDs in codes based on dynamic data structures. Next, in Section 4 we discuss the application of the test to the custom made and benchmark programs. In Section 5 we summarize some of the previous works in the topic of data dependences detection in pointer-based codes. Finally, in Section 6 we conclude with the main contributions and ideas for future work.

2 Shape Analysis Framework

The algorithms presented in this paper are designed to analyze programs with dynamic data structures that are connected through pointers defined in languages like C. The programs, before the analysis, have to be preprocessed in order to normalize the pointer statements. It is, each statement dealing with pointers must contain only simple access paths. Thus, in our approach, we will consider six simple instructions that deal with pointers:

```
x = NULL            x = malloc       x = y

x->field = NULL     x->field = y     x = y->field
```

where `x` and `y` are pointer variables and `field` is a field name of a given data structure. More complex pointer instructions can be built upon these simple ones and temporal variables. As a preprocessing tool, we have used and extended the ANTLR framework [16] which allow us to automatically normalize the C codes before the shape analysis.

Basically, our analysis is based on approximating by graphs (which we name as *Reference Shape Graphs, RSGs*) all possible memory configurations that can appear after the execution of a statement in the code. By *memory configuration* we mean a collection of dynamic structures. These structures comprise several memory chunks, that we call *memory locations*, which are linked by references. Inside these memory locations there may be several fields (data or pointers to other memory locations). The pointer fields of the data structure are called *selectors*. In Fig. 1 we can see a particular memory configuration which corresponds with a single linked list. Each memory location in the list comprises the `val` data field and the `nxt` selector (or pointer field). In the same figure, we can see the corresponding RSG which capture the essential properties of the memory configuration by a bounded size graph. In this graph, the node $n1$ represent the first memory location of the list, $n2$ all the middle memory locations, and $n3$ the last memory location of the list.

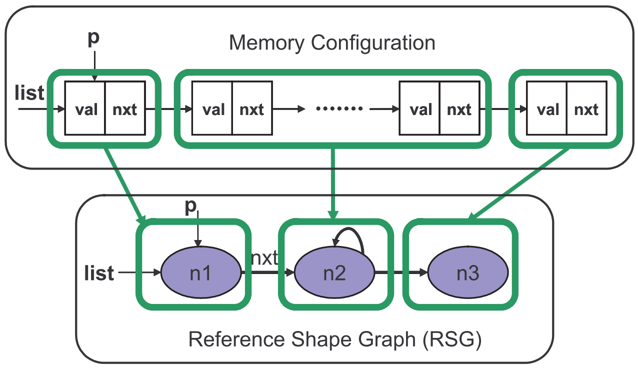
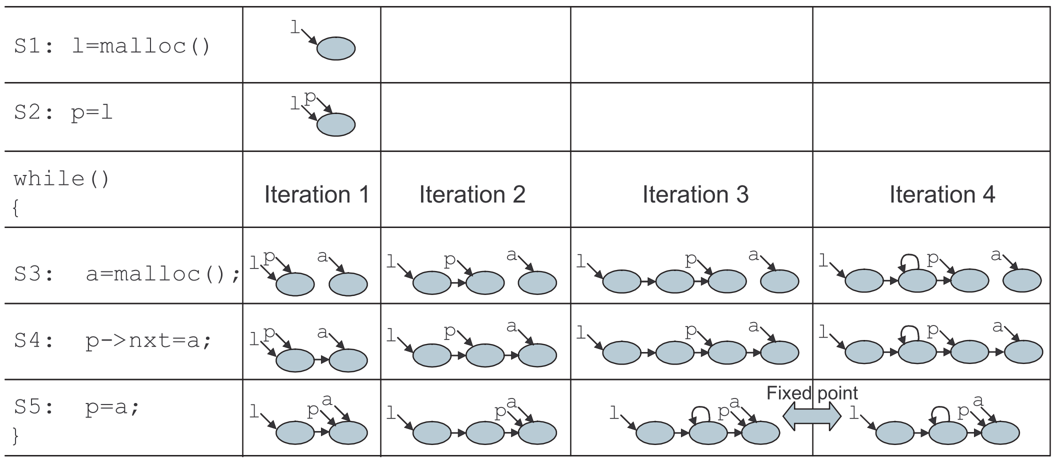


Fig. 1. Working example data structure and the corresponding RSG.

Basically, each RSG is a graph 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 (or summarized) by the same node ($n2$ in our example). These properties are described in [2], but two of them are sketched here because they are necessary in the following sections: (i) the **Share Information**: it 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: (1) *SHARED(n)* states if any of the locations represented by the node n can be referenced by other locations by different selectors (e.g. *SHARED(n2)=FALSE* in the previous figure); (2) *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. For instance, *SHSEL(n2, nxt)= FALSE* captures the fact that following selector *nxt* you always reach a different memory location; and (ii) the **Touch Information**: it is taken into account only inside loop bodies and it helps to mark the memory locations that have been visited. This property avoids the summarization of already visited locations with non-visited ones. We will see how the touch information is the key tool that allows us to automatically annotate the nodes of the data structure which are written and/or read by the pointer statements inside loops.

Each statement of the code may have associated a set of RSGs, in order to represent all the possible memory configuration at each particular program point. In order to generate the set of RSGs associated with each statement (or in other words, to move from the "memory domain" to the "graph domain" in Fig. 1), an **abstract interpretation** of the program statements over the graphs is carried out. Basically, each program statement transforms the graphs to reflect the changes in memory configurations derived from the statement execution. The **abstract semantic** of each statement states how the analysis of this statement must transform the graphs [2]. The abstract interpretation is carried out iteratively for each statement until we reach a fixed point in which the resulting RSGs associated with the statement does not change any more. All this can be illustrated by the example of Fig. 2, where we can see how the statements of the code which builds a single linked list are symbolically executed until a fixed point is reached.



```
S1: l=malloc()
S2: p=l
while()
{
S3:   a=malloc();
S4:   p->nxt=a;
S5:   p=a;
}
```

Fig. 2. Building a RSG for each statement of an example code.

3 Loop-Carried Dependences Detection

As we have mentioned, we focus on detecting the presence of LCDs on loops that traverse heap-based dynamic data structures. Two statements in a loop induce a LCD, if a memory location accessed by one statement in a given iteration, is accessed by the other statement in a future iteration, with one of the accesses being a write access.

The new method tries to identify if there is any LCD in the loop following the algorithm that we outline in Fig. 3. Let's recall that the programs have been normalized such that the statements dealing with pointers contain only simple access paths. Let's assume that statements in the loop have been labeled (S1, S2, ..., S_i, ...). The set of the loop body simple statements (named SIMPLESTMT) is the input to this algorithm.

As we see in Fig. 3, the new algorithm can be divided into four steps. Let's review them in more detail.

3.1 Step 1: Attaching Dependence Touch directives

In the first step, the algorithm examines the simple statements of the loop body and it annotates some of them with a new directive, the Dependence Touch directive, `DepTouch`. More precisely, only a simple pointer statement,

```
fun LCDs_Detection (SIMPLESTMT)
1. ∀ $S_i$ ∈ SIMPLESTMT that accesses the heap
       Attach($S_i$, DepTouch(AccPointer,AccAttS$_i$,AccField));
2. [DEPGROUP,ACCESSPAIRSGROUP] = Create_Dependence_Groups(DEPTOUCH);
3. ACCESSPAIRSGROUP = Shape_Analysis(SIMPLESTMT,DEPTOUCH,
             DEPGROUP,ACCESSPAIRSGROUP);
4. DEP = LCD_Test(ACCESSPAIRSGROUP);
   if DEP == ∅ then
       return(NoLCD); /* no data dependence detected */
   else
       return(DEP); /* return all the detected data dependences */
   endif;
end
```

Fig. 3. The new data dependences detection algorithm.

S_i, that access the heap inside the loop, is annotated with the directive. The directive summarizes the statement information that is relevant to the next steps of our algorithm.

Definition 3.1 *A* **Dependence Touch directive** *is defined as* `DepTouch(AccPointer, AccAttrS`$_i$`, AccField)`*. It comprises three important pieces of information regarding the access to the heap in statement S_i: i) the access pointer, AccPointer, the access attribute, AccAttrS$_i$, and the access field, AccField.* □

Definition 3.2 *The* ***access pointer****, AccPointer: is a stack declared pointer which accesses the heap in the context of a statement.* □

Definition 3.3 *The* ***access attribute****, AccAttS$_i$: identifies the type of access of the access pointer in a S_i statement. This access attribute can take one of the values: ReadS$_i$, WriteS$_i$.* □

Definition 3.4 *The* ***access field****, AccField: is the field of the data structure pointed to by the access pointer which is read or written in the context of a statement.* □

For instance, an `S1: aux = p->nxt` statement should be annotated with `DepTouch(p, ReadS1, nxt)`, whereas the `S4: aux3->val = tmp` statement should be annotated with `DepTouch(aux3, WriteS4, val)`.

3.2 Step 2: Creating Dependence and Access Pairs Groups

In this step, the algorithm creates some auxiliary structures related to the data dependences detection: the Dependence Groups and the Access Pairs Groups.

Definition 3.5 *Let g denote the name of an access field contained in any of the Dependence Touch directives of the loop. A* **Dependence Group** *related to this access field, $DepGroup_g$, can be created. This Dependence Group is a set of access attributes fulfilling two conditions: i) the access attributes belong to Dependence Touchs in which the access field is g, being the access pointers of the same data type, and ii) at least one of these access attributes is a $WriteS_i$.* □

In other words, a $DepGroup_g$ annotates the type of access (read/write) of a set of statements in the loop that access the heap through pointers of the same data type and that may potentially lead to a LCD. A LCD could happen if: i) the analyzed statement makes a write access (WriteS$_i$) or ii) there are other statements accessing the same field (g) and one of the accesses is a write (WriteS$_j$). We outline in Fig. 4 the function `Create_Dependence_Groups`. It creates Dependence Groups, using as an input the set of Dependence Touch directives, DEPTOUCH. Note that it is possible to create a Dependence Group with just one WriteS$_i$ attribute. This Dependence Group will help us check the output dependences for the execution of S_i in different loop iterations. The set of all the Dependence Groups is named DEPGROUP.

As we see in Fig. 4, associated with each $DepGroup_g$, the algorithm initializes a set called $AccessPairsGroup_g$. This set is initially empty but during the next step of the analysis process it could be filled with the pairs named access pairs.

Definition 3.6 *An* **access pair** *is a tupla which comprises two ordered access* *attributes which belong to the same Dependence Group.* □

For instance, a $DepGroup_g = \{\texttt{ReadS}_i, \texttt{WriteS}_j, \texttt{WriteS}_k\}$ is a Dependence Group that contains three access attributes related to three statements in the loop (S_i, S_j and S_k) such as all of the accesses are to the same field (g). At the end of the step 3 of the algorithm, the associated $AccessPairsGroup_g$ could comprise the pair <ReadS_i,WriteS_j>. That would mean that, during the analysis, the same field, g, of the same memory location may have been first read by statement S_i and later written by statement S_j, clearly leading to an anti-dependence. Therefore, the order inside each access pairs is significant for the sake of discriminating between flow, anti or output dependences. The set of all *AccessPairsGroup*'s is named ACCESSPAIRSGROUP. As we see in Fig. 4, the outputs of the `Create_Dependence_Groups` function are the DEPGROUP set and the initial ACCESSPAIRSGROUP set.

```
fun Create_Dependence_Groups(DEPTOUCH)
 DEPGROUP = $\emptyset$; ACCESSPAIRSGROUP = $\emptyset$;
 $\forall$ DepTouch(AccPointer,AccAttS$_i$,g) $\in$ DEPTOUCH
    if [(AccAttS$_i$ == WriteS$_i$) or
    $\exists$ DepTouch(AccPointer',AccAttS$_j$,g) $\in$ DEPTOUCH, being $j \neq i$ /
    (TYPE(AccPointer) == TYPE(AccPointer')) and ( AccAttS$_j$ == WriteS$_j$)] then
       if $\nexists$ $DepGroup_g$ $\in$ DEPGROUP then
          $DepGroup_g$ = {AccAttS$_i$}; DEPGROUP = DEPGROUP $\cup$ $DepGroup_g$;
       else
          $DepGroup_g$ = $DepGroup_g$ $\cup$ {AccAttS$_i$};
       endif;
    endif;
 $\forall$ $DepGroup_g$ $\in$ DEPGROUP
    $AccessPairsGroup_g$ = $\emptyset$ ;
    ACCESSPAIRSGROUP = ACCESSPAIRSGROUP $\cup$ $AccessPairsGroup_g$ ;
return(DEPGROUP,ACCESSPAIRSGROUP);
```

Fig. 4. `Create_Dependence_Groups` function.

3.3 Step 3: Running the Shape Analysis

In this step, we run the shape analysis tool. For it, the algorithm calls the `Shape_Analysis` function whose inputs are the set of simple statements SIM-

PLESTMT, the set of Dependence Touch directives, DEPTOUCH, the set of Dependence Groups, DEPGROUP and the initial ACCESSPAIRSGROUP set. The output of this function is the final ACCESSPAIRSGROUP set. The `Shape_Analysis` function is based in our shape analysis framework, that is described in detail in [2] and briefly introduced in Section 2. In Fig. 5 we outline the necessary extension to our shape analysis presented in [2] in order to deal with the data dependence analysis, which is the main focus in this paper.

Here, we will describe the main features of the shape analysis related to the LCD test. Let's recall that our shape analysis is able to precisely identify, at compile time, the memory locations that are going to be pointed to by the pointers of the loop. Basically, the task of the shape analysis is to symbolically execute each statement locating the access pointer of the statement at the corresponding memory location (or node) and updating the graphs that represent the possible memory configurations. At the same time, with the information provided by the `DepTouch` directive, the node pointed to by the access pointer of the statement, is "*touched*". This means, that the memory location is going to be marked with the access attribute of the corresponding `DepTouch` directive. In that way, we annotate in the memory location, that a given statement has read or written in a given field comprised in the location.

```
fun Shape_Analysis(SIMPLESTMT, DEPTOUCH, DEPGROUP, ACCESSPAIRSGROUP)
 ...
 ∀ $S_j$ ∈ SIMPLESTMT
    ...
    if DepTouch(AccPointer,AccAttS$_j$,AccField) attached to $S_j$ then
       $AccessPairsGroup_g$ = TOUCH_Updating(TOUCH$_n$, AccAttS$_j$,
            $DepGroup_g$, $AccessPairsGroup_g$);
    endif;
    ...
return(ACCESSPAIRSGROUP);
```

Fig. 5. `Shape_Analysis` function extension.

But let's see more precisely how the `Shape_Analysis` function works. The simple statements of the loop body are executed according to the program control flow, and each execution takes the graphs from the previous statement and modifies it (producing a new set of graphs). On the other hand, each node n of a RSG graph has a set associated with it, named Touch Set and this set can be modified during the execution of a statement.

Definition 3.7 *Let n be a node of a RSG graph. The **Touch Set** associated with it, $TOUCH_n$, is a set that contains the access attributes from the* `DepTouch` *directives of the statements that have touched the node.* □

Let's see how the "touch of a node" takes place. When a statement S_j, belonging to the analyzed loop, is symbolically executed the access pointer of the statement, `AccPointer`, will be made to point to a node, n. In our analysis, a node pointed to by a pointer always represents a single memory location. That node n has associated its corresponding $TOUCH_n$ set. If the statement S_j is annotated with a `DepTouch` directive then that directive is also interpreted by the analyzer leading to the updating of the $TOUCH_n$ set.

This TOUCH set updating process can be formalized as follows. Let be `DepTouch(AccPointer,AccAttS`$_j$`,AccField)` the Dependence Touch directive attached to sentence S_j. Let's assume that AccAttS$_j$ belongs to a Dependence Group, $DepGroup_g$. Let n be the RSG node pointed to by the access pointer, `AccPointer`, in the symbolic execution of the statement S_j. Let be $\{AccAttS_k\}$ the set of access attributes which belongs to the $TOUCH_n$ set, where k represents all the statements S_k, which have previously touched the node. $TOUCH_n$ could be an empty set. Then, when this node is going to be touched by the above mentioned `DepTouch` directive, the updating process that we show in Fig. 6 takes place.

As we note in Fig. 6, if the $TOUCH_n$ set was originally empty we just append the new access attribute AccAttS$_j$ of the `DepTouch` directive. This situation happens when the node n is visited (touched) for the first time. However, if the TOUCH set does already contain other access attributes, $\{AccAttS_k\}$, it is because that node has been visited by other statements. Let's recall that the access attributes that we are using to touch the nodes represent information about the type of access of the statements that have

```
fun TOUCH_Updating(TOUCH$_n$, AccAttS$_j$, $DepGroup_g$, $AccessPairsGroup_g$)
 if TOUCH$_n$ == $\emptyset$ then /* The Touch set was originally empty */
    TOUCH$_n$ = $\{AccAttS_j\}$; /* just append the new access attribute */
 else /* The Touch set was not empty */
    $AccessPairsGroup_g$ = AccessPairsGroup_Updating(TOUCH$_n$, AccAttS$_j$,
       $DepGroup_g$, $AccessPairsGroup_g$);
          /* update the access pairs group set */
    TOUCH$_n$ = TOUCH$_n$ $\cup$ $\{AccAttS_j\}$; /* append the new access attribute */
 endif;
return($AccessPairsGroup_g$);
fun AccessPairsGroup_Updating(TOUCH$_n$, AccAttS$_j$, $DepGroup_g$, $AccessPairsGroup_g$)
 $\forall$ AccAttS$_k$ $\in$ TOUCH$_n$
    if AccAttS$_k$ $\in$ $DepGroup_g$ then /* AccAttS$_k$ and AccAttS$_j$ $\in$ $DepGroup_g$ */
       $AccessPairsGroup_g$ = $AccessPairsGroup_g$ $\cup$ {<AccAttS$_k$,AccAttS$_j$>};
          /* A new ordered pair is appended */
    endif;
return($AccessPairsGroup_g$);
```

Fig. 6. TOUCH and AccessPairsGroup updating functions.

previously visited the node. In this case, two actions take place: first, an updating of the $AccessPairsGroup_g$ associated with the $DepGroup_g$ to which the new access attribute AccAttS$_j$ belongs, takes place; secondly, the access attribute AccAttS$_j$ is appended to the TOUCH set of the node, $TOUCH_n = TOUCH_n \cup \{AccAttS_j\}$.

The algorithm for updating the $AccessPairsGroup_g$ is shown in Fig. 6. We had defined the $DepGroup_g$ (def. 3.5) to track the accesses (to the same field g) of a set of statements that may potentially lead to a LCD. In fact, the associated $AccessPairsGroup_g$ is updated when a LCD appears. In the algorithm of Fig. 6, we check all the access attributes of the statements that have touched previously the node n. If there is any access attribute, AccAttS$_k$ which belongs to the same $DepGroup_g$ that AccAttS$_j$ (the current statement which is being executed), then a new access pair is appended to the $AccessPairsGroup_g$. The new pair is an ordered pair <AccAttS$_k$, AccAttS$_j$> which indicates that the memory location represented by node n has been first accessed by statement S_k and later by statement S_j, being S_k and S_j two statements associated with the same Dependence group, therefore a conflict may occur. Note that in the implementation of an $AccessPairsGroup_g$ there will be no redundancies in the sense that a given access pair can not be stored twice in the group.

3.4 Step 4: Checking the data dependences

In the last step, the new `LCD_Test` function will check each one of the previously generated $AccessPairsGroup_g$, which belong to the ACCESSPAIRS-GROUP set. The input of this function is precisely the ACCESSPAIRS-GROUP set generated in the previous step, and the output is the set DEP, which contains all the LCDs detected in all the Access Pairs Groups. The function is detailed in the code of Fig. 7.

Definition 3.8 *The* **set of LCDs** *detected in an Access Pairs Group, AccessPairsGroup$_g$, is named as Dep$_g$. It can be an empty set or it can contain any combination of the following data dependence types:* `FlowDep` *(i.e. Flow dependence),* `AntiDep` *(i.e. Anti-dependence),* `OutDep` *(i.e. Output dependence).* □

Theorem 3.1 *Let AccessPairsGroup$_g$ be the Access Pairs Group associated with the Dependence Group DepGroup$_g$. If, after the shape analysis (step 3), AccessPairsGroup$_g$ is empty, then the statements associated with DepGroup$_g$ do not provoke any LCD.* ■

Proof. The shape analysis iteratively executes the loop body statements in order, and it finishes when a fixed point is reached. That means that all the nodes that the loop body statements visit, have been touched with the corresponding access attribute and the graphs as well as the TOUCH/ACCESSPAIRSGROUPS sets do not change any more. If an Access Pairs Group, *AccessPairsGroup$_g$*, is empty that is because the statements associated with the corresponding Dependence Group, *DepGroup$_g$*, have not visited the same node (or memory location) during the loop execution, therefore we can safely determine that those statements will not carry out any LCD. □

On the other hand, if after the shape analysis, an $AccessPairsGroup_g$ is not empty, then depending on the pairs comprised by the $AccessPairsGroup_g$, we can raise some of the data dependence patterns provided by Lemma 3.1.

Lemma 3.1 *Let <$AccAttS_i$,$AccAttS_j$> be an access pair belonging to a non empty Access Pairs Group, $AccessPairsGroup_g$. A data dependence is reported in the following cases:*

- *$AccAttS_i$ == $WriteS_i$ and $AccAttS_j$ == $ReadS_j$ $\rightarrow$ a Flow dependence (*`FlowDep`*) between statements S_i and S_j has been detected.*
- *$AccAttS_i$ == $ReadS_i$ and $AccAttS_j$ == $WriteS_j$ $\rightarrow$ an Anti-dependence (*`AntiDep`*) between statements S_i and S_j has been detected.*
- *$AccAttS_i$ == $WriteS_i$ and $AccAttS_j$ == $Write_j$ $\rightarrow$ an Output dependence (*`OutDep`*) between statements S_i and S_j has been detected.*

An special case happens when the access pair <$AccAttS_i$,$AccAttS_i$>, being $AccAttS_i$ == $WriteS_i$, belongs to the $AccessPairsGroup_g$. In this case, an Output dependence (`OutDep`*) between statement S_i and itself is detected.* □

Proof. From the proof of Theorem 3.1, we know that if an Access Pairs Group, $AccessPairsGroup_g$, is not empty at the end of the shape analysis, is because the statements associated with the corresponding Dependence Group, $DepGroup_g$, have visited the same field (g) in at least one node (or memory location) during the loop execution. In other words, a data dependence could happen. From the `AccessPairsGroup_updating` function, we see that the components in an access pairs tupla are inserted in an ordered fashion. Therefore this order can be used to accurately discriminate between flow, anti or output dependences. □

The `LCD_Test` function checks all the data dependence patterns considered in Lemma 3.1, for all the access pairs contained in an $AccessPairsGroup_g$, as we see in Fig. 7. This process generates the set Dep_g which contains the LCDs detected in that Access Pairs Group. We note that this checking must be performed for all the Access Pairs Groups. All the Dep_g sets generated are merged into the set DEP, which represents all the LCDs detected in the loop. Precisely that is the output of function `LCD_Test`.

```
fun LCD_Test(ACCESSPAIRSGROUP)
 DEP = $\emptyset$ ;
 $\forall$ $AccessPairsGroup_g$ $\in$ ACCESSPAIRSGROUP
   $Dep_g$ = $\emptyset$;
   $\forall$ access pair $\in$ $AccessPairsGroup_g$
     case (<AccAttS$_i$,AccAttS$_j$>) of
        (<WriteS$_i$,ReadS$_j$>) $\rightarrow$ $Dep_g$ = $Dep_g$ $\cup$ FlowDep; /* Flow dep. between S$_i$ and S$_j$ */
        (<ReadS$_i$,WriteS$_j$>) $\rightarrow$ $Dep_g$ = $Dep_g$ $\cup$ AntiDep; /* Anti dep. between S$_i$ and S$_j$ */
        (<WriteS$_i$,WriteS$_j$>) $\rightarrow$ $Dep_g$ = $Dep_g$ $\cup$ OutDep; /* Output dep. between S$_i$ and S$_j$ */
     endcase;
     case (<AccAttS$_i$,AccAttS$_i$>) of
        (<WriteS$_i$,WriteS$_i$>) $\rightarrow$ $Dep_g$ = $Dep_g$ $\cup$ OutDep; /* Output dep. between S$_i$ and S$_i$ */
     endcase;
   DEP = DEP $\cup$ $Dep_g$ ;
return(DEP); /* return the detected data dependences */
```

Fig. 7. LCD test function.

When the `LCD_Test` function returns, the algorithm in Fig. 3 just has to verify the content of the DEP set. If DEP is empty, that means that none of the dependence patterns is found for any Dependence Group, therefore the algorithm informs that the loop does not contain LCD dependences (`NoLCD`) due to heap-based pointers. In any other case, the set of all LCD dependences detected in the loop (which is DEP) is returned.

3.5 An example

Let's illustrate via a simple example how the approach works. Fig. 8(a) represents a loop that traverses the data structure of Fig. 1. This is, this loop is going to be executed after the building of the linked list data structure due to the code of Fig. 2. In the loop, the statement `tmp = p->val` reads a memory location that has been written by `p->nxt->val = tmp` in a previous iteration, so there is a LCD between both statements.

```
p = list;
while (p->nxt != NULL)
{
  tmp = p->val;
  p->nxt->val = tmp;
  p = p->nxt;
}
```

(a)

```
p = list;
while (p -> nxt != NULL)
{
S1: tmp = p->val; DepTouch(p, ReadS1, val);
S2: aux = p->nxt; DepTouch(p, ReadS2, nxt);
S3: aux->val = tmp; DepTouch(aux, WriteS3, val);
S4: p = p->nxt; DepTouch(p, ReadS4, nxt);
}
```

(b)

Fig. 8. (a) Loop traversal of a dynamic data structure; (b) Instrumented code used to feed our shape analyzer.

In order to automatically detect this LCD, we use an ANTLR-based preprocessing tool that atomizes the complex pointer expressions into several simple pointer statements which are labeled, as we can see in Fig. 8(b). For instance, the statement `p->nxt->val = tmp;` has been decomposed into two simple statements: S2 and S3. After this step, the SIMPLESTMT set will comprise four simple statements.

Next, by applying the first step of the algorithm to find LCDs, the `DepTouch` directive is attached to each simple statement in the loop that accesses the heap, as we can also appreciate in Fig. 8(b). For example, the statement `S2: aux = p->nxt` has been annotated with the `DepTouch(p, ReadS2, nxt)`, stating that the access pointer is `p`, the access attribute is `ReadS2` (which means that the S2 statement makes a read access to the heap) and finally, that the read access field is `nxt`. This first step of the new method have been also implemented with the help of ANTLR.

Next we move on to the second step in which we point out that statements S1 and S3 in the code example meet the requirements to be associated with a Dependence Group: both of them access the same access field (`val`) with pointers of the same type (`p` and `aux`), being S3 a write access. We will define this Dependence Group as $\mathit{DepGroup}_{val}$={ReadS1, WriteS3}. Besides, the associated $AccessPairsGroup_{val}$ set will be, at this point, empty. Therefore, after this step, DEPGROUP = $\{DepGroup_{val}\}$ and ACCESSPAIRSGROUP = $\{AccessPairsGroup_{val}\}$.

Let's see now how step 3 of the algorithm proceeds. As we have mentioned, Fig. 1 represents the only RSG graph of the RSGs set at the loop entry point. Remember that our analyzer is going to symbolically execute each of the statements of the loop iteratively until a fixed point is reached. This is, all the RSG graphs in the RSGs set associated with the statements will be updated at each symbolic execution and the loop analysis will finish when all the graphs in the RSGs do not change any more.

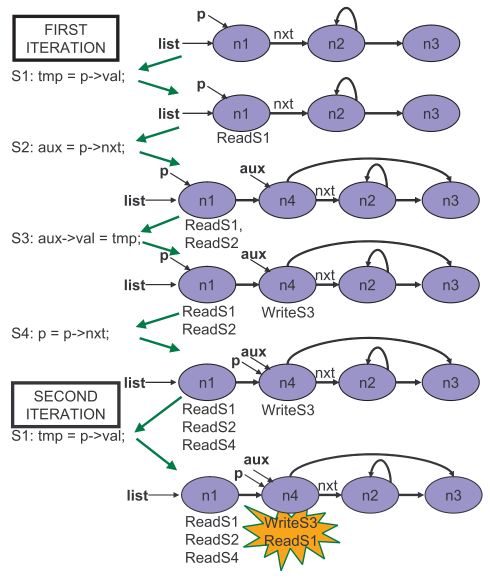


Fig. 9. Initial RSG at the loop entry and the resultant RSG graphs when executing S1, S2, S3 and S4 in the first loop iteration, and when S1 is executed in the second loop iteration. We illustrate the TOUCH$_{n1}$ and TOUCH$_{n4}$ sets.

Now, in the first loop iteration, the statements S1, S2, S3 and S4 are executed by the shape analyzer. The resultant RSG graphs when these statements are symbolically executed, taking into account the attached `DepTouch` directives, are shown in Fig. 9. Executing S1 will produce that the node pointed to by `p` (n_1) is touched by `ReadS1`. When executing S2, `aux = p->nxt` will produce the materialization of a new node (the node n_4), and the node pointed to by `p` will be touched by `ReadS2`. Next, the execution of S3 will touch with a

WriteS3 attribute, the node pointed to by aux (n_4). Finally, the execution of S4 will touch with a ReadS4 attribute the node n_1, and then p will point to node n_4.

In the second loop iteration, when executing S1 over the RSG graph that results from the previous symbolic execution of S4, we find that the nodes pointed to by p (now node n_4) is touched by ReadS1. When touching this node, the TOUCH_Updating function detects that the node has been previously touched because $TOUCH_{n4}$ ={WriteS3}. Since the set is not empty, the function will call to the AccessPairsGroup_Updating function. Now, this function will check each access attribute in the $TOUCH_{n4}$ set, and it will look for a Dependence Group for such access attribute. In the example, WriteS3 is in the $DepGroup_{val}$. In this case, since the new access attribute that is going to touch the node (ReadS1) belongs to the same Dependence Group, a new access pair is appended to the $AccessPairsGroup_{val}$= {<WriteS3, ReadS1>}. This fact is indicating that the same memory location (in this case the field *val* in node n_4) has been reached by a write access from statement S3, followed by a read access from statement S1.

The shape analyzer follows, iteratively, the symbolic execution of statements in the loop until a fixed point is reached. The resultant RSG graph is shown in Fig. 10. We also get at the end of the analysis that $AccessPairsGroup_{val}$= {<WriteS3, ReadS1>}.

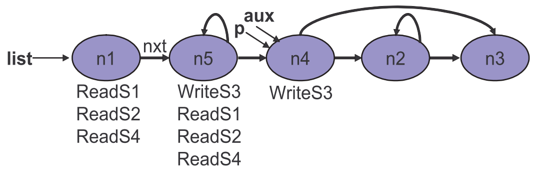


Fig. 10. Resultant RSG when the fixed point is reached. We represent the TOUCH$_{n1}$, TOUCH$_{n4}$ and TOUCH$_{n5}$ sets.

The algorithm applies now the fourth step: the LCD test function (Fig. 7) is called. Initially the DEP set is defined as empty, and then the access pair comprised in the $AccessPairsGroup_{val}$ is checked. The function finds that Dep_{val}=FlowDep. Now, DEP = Dep_{val} = FlowDep. Therefore the LCDs detection function, return only a flow dependence, because the only access pair group, $AccessPairsGroup_{val}$ in the ACCESSPAIRSGROUP set, contains a <WriteS3, ReadS1> pair. As we see, our LCDs detection algorithm accurately captures the LCD that appears in the loop.

4 Experimental results

4.1 Some preliminary results

We have successfully tested our LCDs detection algorithm against some sample codes: (a) a code with a traversal of a DAG data structure; (b) a code with cyclic access in a Cyclic data structure; and (c) a code with conditional cyclic access in a Cyclic data structure. The analysis took less than 1 second for these programs altogether. The reader can found the details in [13]. The goal of these preliminary experiments was to prove the accuracy of our method in the detection of data dependences in codes that traverse (and/or create) complex data structures. Precisely, they are the type of codes where previous research projects could not accurately detect the data dependences. We will discuss this issue in more detail in Section 5.

4.2 Results in real codes

After having applied the new LCDs detection algorithm to the sample codes with successful results, now we put our analysis to work with more realistic codes. For this purpose we have considered several cases of study: i) sparse matrix by sparse vector product (`matrix-vector`); ii) sparse matrix by sparse matrix product (`matrix-matrix`); iii) `mst` code from the Olden suite [1]; iv) `em3d` from the Olden suite too, and v) `twolf` code from the SPEC CPU2000 suite [15]. The first two codes are custom-made programs that represent the kernel of typical real world applications which deal with dynamic data structures and they are available through our website[1].

[1] http://www.ac.uma.es/∼asenjo/research/codes.html

When analyzing the codes looking for LCDs, we do not consider a whole-program approach due to the complexity and number of graphs that our shape analysis approach may generate. Instead we focus our attention on certain loops that carry a significant amount of execution time. For the custom-made programs, i.e., the sparse `matrix-vector` and `matrix-matrix` product codes, we chose the outer loop from the respective nesting that compute the product. But for `mst`, `em3d` and `twolf` profiling information was considered. For instance, in the `twolf` code, the main loop of function `new_dbox`, which represents around the 38% of execution time was studied. In this code, from profiling again, we find that another 8% of the execution time is spent in the function `add_penal`, which we analyze too. In the `mst` code, a loop of the `AddEdges` function, which spends more than 85% of the execution time is selected. And, finally in the `em3d` code, again through profiling, we select the most computationally expensive loop.

Once we select a loop to study, the program is preprocessed before the analysis can be performed. During this process we go through the code finding the statements needed to create the data structures that are going to be traversed/modified in the loop of interest. In this preprocess, function calls are inlined, due to the lack of interprocedural support within our framework at this point. This is not a big issue because we are focusing on no more than a handful of function calls, since only the needed code for structure creation and traversal of study is analyzed. Once we have the proper code isolated, we run it through our ANTLR-based preprocessing tool that performs statements decomposition, nullifies "dead" pointers, asserts pointer values when known, add the `DepTouch` directives to the loop statements that may lead to LCDs and creates the calls to the shape analysis tool. Let's recall that the new LCDs detection test is directly embedded into the shape analysis process, so it should not be considered a later pass but an extension of the shape analysis itself. The test returns the access pairs that have been created during abstract interpretation of the statements in the instrumented program, and their corresponding dependence pattern (`FlowDep`, `AntiDep` or `OutDep`). In the case that no dependence is found, then the test returns `NoDep`.

Next, Table 1 lists the codes being analyzed along with a short description of them, as well as a short presentation of the data structures traversed/created in each one of the selected loops, the number of statements resulting after all the preprocessing (**No. St** column), the number of `DepTouch`'s considered for each loop (**No. DT** column) and finally the dependences detected (**Dep** column) after applying the new LCDs detection test. Also, Fig. 11 depicts the main data structures traversed/created in the loops of study.

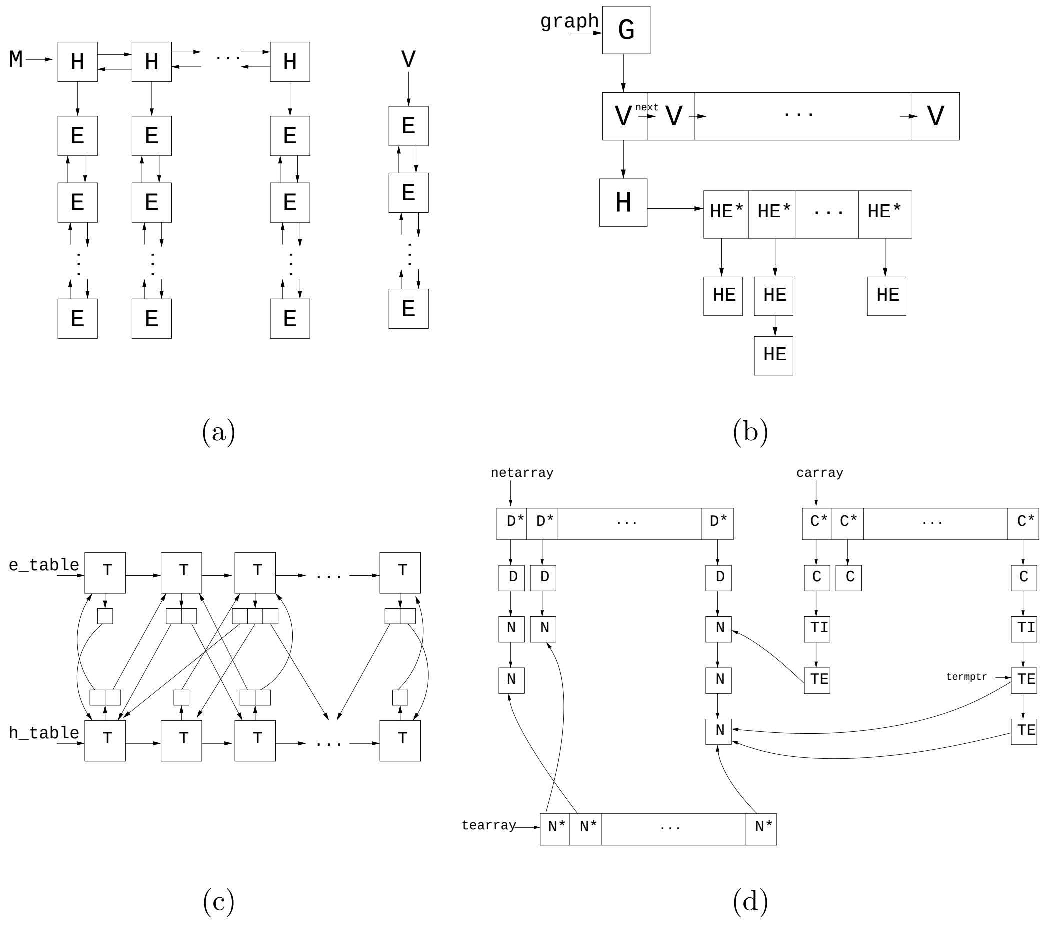


Fig. 11. Main data structures: (a) `matrix-vector`; (b) `mst` ; (c) `em3d`; (d) `twolf`.

Program	**Description**	**Data Structures**	**No. St**	**No. DT**	**Dep**
`matrix-vector`	sparse matrix vector product	matrix: double-linked list of header elements (one by row) each of whom points to another double-linked list; vectors (both input and output): double-linked lists of non-zero elements. The output vector is created during the product	120	6	`NoDep`
`matrix-matrix`	sparse matrix matrix product	input matrices: as explained above; the output matrix is created inside the loop	146	3	`NoDep`
`mst` (`AddEdges`)	Construction of minimum spanning trees	An array of vertexes, each of them points to a hash node and each hash node points to an array of pointers of hash entries	62	3	`FlowDep`, `AntiDep`, `OutDep`
`em3d`	Models the propagation of electromagnetic waves	Two lists of nodes: one for the electrical field and another for the magnetic field; Each node links to the following element in the list; Besides, each node is linked to the values of some nodes from the other list	75	3	`AntiDep*`
`twolf` (`new_dbox`)	It determines the placement and global connections of groups of transistors for the production of microchips	Three interconnected structures: two are array of pointers whose elements point to single-linked lists of nodes	154	2	`FlowDep`, `AntiDep`, `OutDep`
`twolf` (`add_penal`)	It is responsible for updating the values of penalties in the data structure	A matrix of nodes accessed by following a couple of pointer arrays	49	9	`FlowDep*`, `AntiDep*`

Table 1

Benchmarks characteristics. The * means distance 0 dependences (No LCD).

In all the cases, the new data dependence detection algorithm found, accurately, the dependences which may rise in the loops. But after close examination, we discover that some of the access pairs that build our method and that are identified in the LCD test function (see Fig. 7) as a flow, anti or output dependence, are not LCDs, properly speaking. Such access pairs represent data dependence with *distance 0* [12], i.e. dependences that arise between two statements of the loop in the current iteration. In other words, the distance 0 dependences are not loop carried data dependences and they should not prevent the loop from being parallelized as well as another loop transformations that avoid reordering such statements in the body. Thus, it could be interesting detecting when they appear. In fact, there are three cases of distance 0 dependences in the benchmarks, and they are marked with a * in Table 1: the access pair which produce an anti dependence in the `em3d` loop and the access pairs that generate the flow and the anti dependences in the `twolf (add_penal)` function.

Our algorithm can easily be extended to mark the access pairs that present a distance 0 dependence. For it, the shape analysis tool must maintain a symbolic counter for each loop that surrounds the loop body statements being analyzed. Each counter is incremented each time that the analysis reach the header of the corresponding loop. In the `TOUCH_Updating` function, each time that an access attribute is appended to the appropriate TOUCH set, it will be annotated with the corresponding iteration vector [12]. That vector contains one entry for each counter of the loops surrounding the statement, and it denotes the symbolic iteration instance at which the access attribute of the statement touches the node. Next, when the `AccessPairsGroup_Updating` function detects that a new access pair appears, it computes the distance vector [12] from the iteration vectors of the two access attributes comprised in the pair and it appends this information to the pair. When a 0 appears in a k-th component of a distance vector, then it means that, for such a pair, a distance 0 dependence appears in the k-th loop that surround the loop body statements, therefore that dependence is not a LCD in the k-th loop.

Once the accuracy of the new LCDs detection algorithm, in the context of real codes, have been tested, we are interested in studying the behavior of the compilation algorithms. For it, we conducted two set of experiments for each code: in the first set, we run our LCDs detection algorithm based in shape analysis and we obtained the data dependence information previously reported. In addition, we collected more information that is summarized in Table 2. As we were interested in measuring how much complexity was added in the analysis due to the LCDs detection algorithm, we conducted a second set of experiments. In this set of experiments, we carried out only the shape analysis, i.e. without the LCDs detection. Table 3 refers to the gathered data for this second experiment. For each program we can find the analysis time for the experiments (including the underlying shape analysis necessary to build the structures) in the **Time**[2] column. Follows the number of symbolic iterations carried out by the shape analyzer to reach a fixed point in the **No. It.** column. **No. Graphs** shows the number of RSG graphs created in the course of the analysis, including temporal graphs that may appear during a statement execution. Finally, the **Graphs/Stmt.** column indicates the average number of graphs generated by each analyzed statement.

Although not showed in the tables, we collected too, for both sets of experiments, the size of the log files, which give us an indication of the memory used to store the generated graphs. We found that the size of such log files ranges from 250 KBytes to 2.1 MBytes, that we think is quite reasonable. Next, we discuss the results from both experiments in more detail.

[2] These times were obtained in a Pentium IV 2.8GHz with 1 GBytes main memory. Similar times were obtained with a 512 Mbytes main memory configuration, which seems to indicate that the analyses are not a memory bound processes.

4.3 Discussion of the results

From Table 2 we see that the analysis times for the LCDs detection range from seconds to several minutes. Basically, these differences depend on the complexity of the data structures and the traversal. For instance, the time for the code `twolf (add_penal)` is the smallest because the data structure traversed in the loop of study, is the smallest one, and in addition, its connections are very simple. Therefore the number of graphs generated during the analysis and the no. of iterations to reach the fixed point are the lowest ones, respectively. However it is the loop where more DepTouch directives are appended. On the other hand, the time for the code `twolf (new_dbox)` is significantly higher, as well as the no. of iterations and graphs, because now the traversed structure is much more complex. Another loop that traverses a very complex structure is the one selected in the `em3d` code, but although the no. of iterations to reach the fixed point is high, surprisingly the number of generated RSGs is modest. The worst time corresponds to the loop in the `matrix-matrix` code, and, as it could be expected, the no. of iterations to reach the fixed point and the number of graphs generated are the highest ones.

Program	**Time**	**No. It.**	**No. Graphs**	**Graphs/Stmt.**
`matrix-vector`	1 min 47 sec	315	170450	1420.4
`matrix-matrix`	1 h 34 min	814	1421076	9733.4
`mst`	0.7 sec	70	3314	53.5
`em3d`	3.7 sec	508	12908	172.1
`twolf (new_dbox)`	5 min 54 sec	149	112803	732.5
`twolf (add_penal)`	0.1 sec	12	363	7.4

Table 2

Time and other measures for the codes when running the dependence test based on shape analysis.

In general, the number of iterations to reach the fixed point seems related to the control flow complexity of the loop being analyzed. For instance, the `matrix-matrix` is a 4-level loop nesting, whereas the `matrix-vector` and `em3d` are 3-level and 2-level loop nesting, respectively. However, the most determining factor in the analysis times seems to be the number of generated graphs. In this case, the outstanding case is the `matrix-matrix` code, for which 1,421,076 graphs are generated. But not only the no. of graphs is the key to explain high analysis times. For instance, the no. of graphs generated in the `twolf (new_dbox)` analysis is smaller than in the `matrix-vector` code, but the times are higher. What happens is that the graphs in the `twolf (new_dbox)` loop are more complex: the nodes are more interconnected with each other, and in addition, each node has to maintain more information.

The results shown in Table 3 give us a glimpse of the analysis times, as well as the number of symbolic iterations to reach a fixed point and the number of RSG graphs created in the course of the analysis, when the LCDs detection algorithm is not enabled. As we mentioned before, these results are purely due to the underlying shape analysis framework, and are relevant because decouple the dependence detection algorithm from the shape analysis. From the table we see that the analysis times for the shape analysis range from seconds to some minutes. In general, the analysis times are smaller when compared to the analysis times with the dependence detection algorithm enabled. Now, again, the most determining factor in the shape analysis times seems to be the higher number of generated graphs. In this case, the `matrix-matrix` and `twolf (new_dbox)` codes are the ones which higher number of graphs and the worst times.

In general, the number of generated graphs is much smaller when the dependence detection algorithm is disabled, because now we have removed the DepTouch directives from the code, which means that nodes will not be touched with the read/write attributes, in other words, the probability of summarization of nodes increases, and as a consequence a lesser number of possible configurations should be generated. But in two particular cases, `twolf (new_dbox)` and `mst` the number of generated graphs when the dependence detection algorithm is disabled/enabled are similar (in fact, in the mst code, the number of graphs is slightly higher in the disabled dependence detection version). As a consequence, the analysis times, on these codes, when the dependence detection algorithm is disabled/enabled, are similar too. These results seem to indicate that in simple loops (i.e., those with a low level of nesting) that traverse/create simple data structures (mst case) or even simple loops which traverse complex and heavily interconnected data structures (twolf (new_dbox) case), the data dependence detection algorithm does not provoke a significant increment in the analysis times. However, loops that traverse/create complex data structures in the context of a deep loop nesting, are heavily affected when the LCDs dependence detection algorithm is enabled. This is the case of the matrix-vector and the matrix-matrix loops, where we see a significant increment in the number of generated graphs in the enabled dependence detection versions.

Program	**Time**	**No. It.**	**No. Graphs**	**Graphs/Stmt.**
matrix-vector	9.1 sec	225	30488	254
matrix-matrix	12 min 24.6 sec	588	409026	2801.5
mst	0.7 sec	57	3326	53.6
em3d	2.5 sec	438	8067	107.6
twolf new_dbox)	5 min 13 sec	146	102535	665.8
twolf (add_penal)	0.03 sec	10	151	3.1

Table 3

Time and other measures for the codes when running only the shape analysis (i.e., without dependence detection).

As we see, the main limitation of the current algorithm presented in this paper is related, mainly, to the underlying shape analysis. This limitation is, as in the case of most existing shape abstractions, that the number of possible graphs generated by our analysis, has an exponential worst-case complexity. However, in the analyzed codes, except in the matrix-matrix product, we have found that the number and size of the graphs is reasonably bounded. Currently, we are working in a new shape analysis abstraction in which the number of generated graphs will be greatly reduced, because instead of a set of RSG graphs per statement, we will maintain only one graph per statement. The key of the new shape abstraction should be to represent the same information but in a more compact way. However, the new RSG could be more complex than the RSGs obtained through the techniques presented in this paper. A careful research will be conducted in this topic to investigate the impact of the new shape analysis abstraction in the data dependence detection accuracy and analysis times.

Overall, we consider that the results are quite encouraging. They suggest that our techniques based on shape analysis can provide very accurate data dependence information in the context of real codes which traverse and create generic and complex heap-based data structures at reasonable analysis times. In short, the results have been promising, but at this stage, we think that this kind of analysis is suitable for analyzing only selected parts of the code, for instance the computationally most expensive loops, as we have done in these experiments. One key aspect of the new method is that can analyze loops where creation and modification of data structures takes place (as in the `matrix-vector`, `matrix-matrix` and `mst` codes). In addition, our approach conveniently distinguishes between flow, anti and output dependences, which is basic for doing data-cache optimizations. Let's note that all these aspects are left out in every other approach we know. Let's see this issue next.

5 Related work

Some of the previous works on data dependences detection on dynamic data structures based codes, combine dependence analysis techniques with pointer analysis [5], [6], [11], [7], [4], [8]. Horwitz et al. [6] developed an algorithm to determine dependences by detecting interferences in reaching stores. Larus and Hilfinger [11] propose to identify access conflicts on alias graphs using path expressions to name locations. Hendren and Nicolau [5] use path matrices to record connection information among pointers and present a technique to recognize interferences between computations for programs with acyclic structures. The focus of these techniques is on identifying dependences at the function-call level and they do not consider the detection in the loop context, which is the focus in our approach.

More recently, some authors [4], [8], [9] have proposed data dependences detection algorithms based on shape analysis in the context of loops that traverse dynamic data structures, and these approaches are more related to our work. For instance, Ghiya and Hendren [4] proposed a test for identifying LCDs that relies on the shape of the data structure being traversed (Tree, DAG or Cycle), as well as on the computation of the access paths for the pointers in the statements being analyzed. In short, their approach identifies dependences in programs with Tree-like data structure or loops that traverse DAG/Cycle structures that have been asserted by the programmer as acyclic and where the access paths do not contain pointer fields. Note that the manual assertion of loops traversing DAG or cyclic data structures is a must in order to enable any automatic detection of LCDs. Another limitation of this approach is that data structures must remain static during the data traversal inside the analyzed loops. For instance, this approach could not handle the loops that we have studied in the `matrix-vector`, `matrix-matrix`, `mst`, `em3d` or `twolf (new_dbox)` codes.

In order to solve some of the previous limitations, Hwang and Saltz proposed a new technique to identify LCDs in programs that traverse cyclic data structures [8], [9]. This approach automatically identifies acyclic traversal patterns even in cyclic (Cycle) structures. For this purpose, the compilation algorithm isolates the traversal patterns from the overall data structure, and next, it deduces the shape of these traversal patterns (Tree, DAG or Cycle). Once they have extracted the traversal-pattern shape information, dependence analysis is applied to detect LCDs. Summarizing, their technique identifies LCDs in programs that navigate cyclic data structures in a “clean” tree-like traverse. On the other hand, their analysis can overestimate the shape of the traverse when the data structure is modified along the traverse or it has been built inserting nodes between existing ones. In these situations, the shape algorithm detect DAG or Cycle traversal patterns, in which case dependence is reported. Another limitation of this approach is that can not accurately distinguish among flow, anti and output dependences.

We differ from previous works in that our technique let us annotate the memory locations reached by each heap-directed pointer with read/write information. This feature let us analyze quite accurately loops that traverse and create generic heap-based dynamic data structures. The new algorithm is able to identify accurately the dependences that appears even in loops that navigate (and create) cyclic structures in traversals that contain cycles, as we have mentioned in Section 4.1. Besides we can successfully discriminate among flow, anti and output dependences. In addition, we have put to work our techniques with real life programs achieving encouraging results, as we have discussed in Sections 4.2 and 4.3.

6 Conclusion and Future Work

We have presented a compilation technique that is able to identify LCDs in programs which work with general pointer-based dynamic data structures. We base our algorithms in a powerful shape analysis framework that let us analyze quite accurately loops that traverse and create heap-based dynamic data structures. The new algorithm is able to precisely identify dependences even in loops that navigate (and create) cyclic structures in traversals that contain cycles. Our main contribution is that we have designed a LCD test that let us extend the scope of applicability to any program that handle any kind of dynamic data structure. Moreover, our dependence test let us discern accurately the type of dependence: flow, anti, output.

We have an implementation of th compilation algorithms and we have conducted several tests that prove it can be useful for real-life programs. However, more work is necessary in order to fully integrate the preprocessing tool for the analysis and provide it with meaningful ways of discarding dispensable information to achieve faster analysis. In addition, although our shape analysis is quite accurate, it still presents some scalability problems when analyzing large programs, which we have overcome focusing our analysis in the most computational expensive loops. In order to address this issue, another line of research consists in developing a general framework which let us conduct demand driven and incremental analysis. These type of analysis would allow us to incorporate the shape analysis (and the dependence detection as well as other optimizations) only to selected parts of the code, to which the relevant shape information have been propagated. As we mentioned before, reducing the number of generated graphs during the analysis is another important topic in our future research. To address it, we are currently working on new shape analysis abstractions which let us compact the information and therefore to reduce the number of generated graphs while they still maintain the same accuracy.

Acknowledgment

This work was supported in part by the Ministry of Education of Spain under contract TIC2003-06623.

References

[1] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in olden. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, July 1995.

[2] F. Corbera, R. Asenjo, and E.L. Zapata. A framework to capture dynamic data structures in pointer-based codes. *Transactions on Parallel and Distributed*

System, 15(2):151–166, 2004.

[3] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *Proc. 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 121–133, San Diego, California, January 1998.

[4] R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in c programs with recursive data strucutures. In *Proc. 1998 International Conference on Compiler Construction*, pages 159–173, March 1998.

[5] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1:35–47, January 1990.

[6] S. Hortwitz, P. Pfeiffer, and T. Repps. Dependence analysis for pointer variables. In *Proc. ACM SIGPLAN'89 Conference on Programming Language Design and Implementation)*, pages 28–40, July 1989.

[7] J. Hummel, L. J. Hendren, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation)*, pages 218–229, June 1994.

[8] Y. S. Hwang and J. Saltz. Identifying parallelism in programs with cyclic graphs. In *Proc. 2000 International Conference on Parallel Processing*, pages 201–208, Toronto, Canada, August 2000.

[9] Y. S. Hwang and J. Saltz. Identifying parallelism in programs with cyclic graphs. *Journal of Parallel and Distributed Computing*, 63(3):337–355, 2003.

[10] N.D. Jones and S.S. Muchnick. *Program flow analysis: theory and applications. Flow analysis and optimization of LISP-like structures*, chapter 4, pages 102–131. Prentice-Hall, 1981.

[11] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. ACM SIGPLAN'88 Conference on Programming Language Design and Implementation)*, pages 21–34, July 1988.

[12] Steven S. Muchnick. *Advanced Compiler Design Implementation.* Morgan Kaufmann, San Francisco, California, 1997.

[13] A. Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plata, and E.L. Zapata. A new dependence test based on shape analysis for pointer-based codes. In *The 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC '04)*, West Lafayette, IN, USA, September 2004.

[14] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proc. 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997.

[15] Standard Performance Evaluation Corporation (SPEC). *SPEC CPU2000 V1.2 Documentation*, 2000. http://www.spec.org/cpu2000/docs/.

[16] T.J.Parr and R.W. Quong. ANTLR: A predicated-LL(k) parser generator. *Journal of Software Practice and Experience*, 25(7):789–810, July 1995.

[17] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, California, June 1995.

List of Tables

List of Figures