A new Dependence Test based on Shape Analysis for Pointer-based Codes*

A. Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plata and E.L. Zapata

Dpt. of Computer Architecture, University of Málaga,
Campus de Teatinos, PB: 4114, E-29080. Málaga, Spain.
{angeles,corbera,asenjo,tineo,oscar,ezapata}@ac.uma.es

Abstract. 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. Our 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. Basically, the novelty is that our approach carries out abstract interpretation of the statements being analyzed, and let us annotate the memory locations reached by each statement with read/write information. This information will be later used in order to find dependences in a very accurate dependence test which we introduce in this paper.

1 Introduction

Optimizing and parallelizing compilers rely upon accurate static disambiguation of memory references, i.e. determining at compiling time if two given memory references always access disjoint memory locations. 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 [12], [10], [2]. 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

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

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 [1].

The approach presented in this paper focus 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 [3], [8], 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 inside a loop. Moreover, as we will see in the next sections, our shape analysis is based on the symbolic execution of the program statements over the graphs that represent the data structure at each program point. In other words, our 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 executes, at compile time, 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 dependence test which we introduce in this paper.

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

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 or C++. The programs have to be normalized in such a way that each statement dealing with pointers contains only simple access paths. This is, we 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. We have used and extended the ANTLR tool [11] in order to automatically normalize and pre-process the C codes before the shape analysis.

Basically, our analysis is based on approximating by graphs (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.

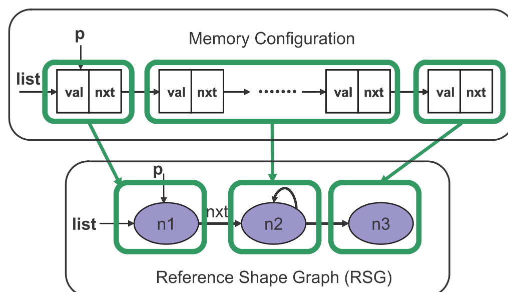

Fig. 1. Working example data structure and the corresponding RSG.

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 [1], but two of them are sketched here because they are necessary in the following sections: (i) the **Share Information** can tell whether at least one of the locations represented by a node is referenced more than once from other memory locations. 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** is taken into account only inside loop bodies to avoid the summarization of already visited locations with non-visited ones. The touch information will be also the key tool in order 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), a **symbolic execution** of the program over the graphs is carried out. 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 [1]. 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.

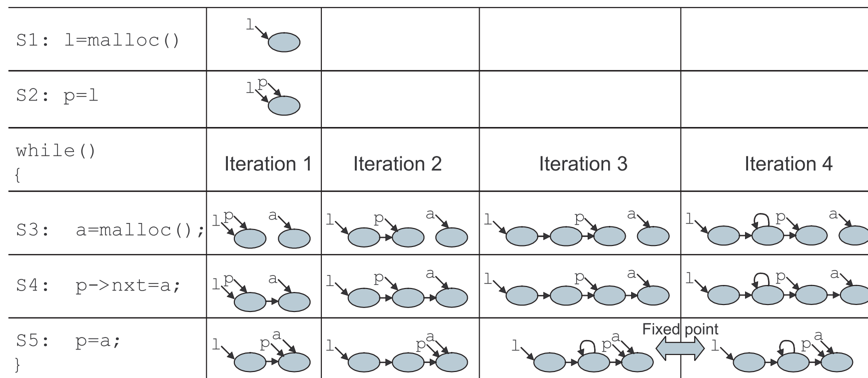

Fig. 2. Building an RSG for each statement of an example code.

3 Loop-Carried Dependence 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.

Our 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 our programs have been normalized such that the statements dealing with pointers contain only simple access paths. Let's assume that statements have been labeled. The set of the loop body simple statements (named SIMPLESTMT) is the input to this algorithm.

```
fun LCDs_Detection (SIMPLESTMT)
1. ∀ S_i ∈ SIMPLESTMT that accesses the heap
        Attach(S_i, DepTouch(AccPointer,AccAttS_i,AccField));
2. DEPGROUP = Create_Dependence_Groups(DEPTOUCH);
    ∀ DepGroup_g ∈ DEPGROUP
        AccessPairsGroup_g = ∅ ;
3. ACCESSPAIRSGROUP = Shape_Analysis(SIMPLESTMT, DEPTOUCH, DEPGROUP);
4. ∀ AccessPairsGroup_g ∈ ACCESSPAIRSGROUP
        Dep_g = LCD_Test(AccessPairsGroup_g);
    if ∀ g, Dep_g == NoDep then
        return(NoLCD);
    else
        return(Dep_g);
    endif;
end
```

Fig. 3. Our dependences detection algorithm.

Summarizing, our algorithm can be divided into the following steps:

1. Only the simple pointer statements, S_i, that access the heap inside the loop are annotated with a **Dependence Touch**, `DepTouch`, directive. 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: is the stack declared pointer which access to the heap in the statement; ii) The **access attribute**, AccAttS_i: identifies the type of access in the statement (ReadS_i or WriteS_i); and iii) The **access field**, AccField: is the field of the data structure pointed to by the access pointer which is read or written. 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)`.
2. The **Dependence Groups**, are created. A Dependence Group, $DepGroup_g$, is a set of access attributes fulfilling two conditions:

– all the access attributes belong to Dependence Touchs with the same access field (g) and access pointers of the same data type, and
– at least one of these access attributes is a WriteS_i.

In other words, a $DepGroup_g$ is related to a set of statements in the loop that may potentially lead to a LCD, which happens if: i) the analyzed statement makes a write access (WriteS_i) or ii) there are other statements accessing to the same field (g) and one of the accesses is a write. 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 to check the output dependences for the execution of S_i in different loop iterations. As we see in Fig. 4 the output of the function is the set of all the Dependence Groups, named DEPGROUP.

```
fun Create_Dependence_Groups(DEPTOUCH)
 DEPGROUP = ∅;
 ∀ DepTouch(AccPointer_i,AccAttS_i,AccField_i) ∈ DEPTOUCH
    if [(AccAttS_i == WriteS_i) or
    ∃ DepTouch(AccPointer_j,AccAttS_j,AccField_j) being j ≠ i /
    (AccField_i == AccField_j) and (TYPE(AccPointer_i) == TYPE(AccPointer_j)) and
    (AccAttS_i == WriteS_i or AccAttS_j == WriteS_j)] then
       g = AccField_i;
       if ∄ DepGroup_g ∈ DEPGROUP then
          DepGroup_g = {AccAttS_i}; DEPGROUP = DEPGROUP ∪ {DepGroup_g};
       else
          DepGroup_g = DepGroup_g ∪ {AccAttS_i};
       endif;
    endif;
return(DEPGROUP);
```

Fig. 4. `Create_Dependence_Groups` function.

Associated with each $DepGroup_g$, our algorithm initializes a set called $AccessPairsGroup_g$ (see Fig. 3). This set is initially empty but during the analysis process it may be filled with the pairs named **access pairs**. An access pair comprises two ordered access attributes. For instance, a $DepGroup_g$ = {ReadS_i, WriteS_j, WriteS_k} with an $AccessPairsGroup_g$ comprising the pair <ReadS_i,WriteS_j> means that during the analysis the same field, g, of the same memory location may have been first read by the statement S_i and then written by statement S_j, clearly leading to an anti-dependence. 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.

3. The shape analyzer is fed with the instrumented code. As we have mentioned, the shape analyzer is described in detail in [1] and briefly introduced in Section 2. However, with regard to the LCD test implementation the most

important idea to emphasize here is that our analyzer is able to precisely identify at compile time the memory locations that are going to be pointed to by the pointers of the code. Basically, the task of the analyzer is to symbolically execute each statement updating the graphs. 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. In this step, our algorithm call to the `Shape_Analysis` function whose inputs are the set of simple statements SIMPLESTMT, the set of DepTouch directives, DEPTOUCH, and the set of Dependence Groups, DEPGROUP. The output of this function is the final set ACCESSPAIRSGROUP. In Fig. 5 we outline the necessary extension to our shape analysis presented in [1] in order to deal with the dependence analysis.

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

Fig. 5. `Shape_Analysis` function extension.

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 a new set of graphs). When a statement S_j, belonging to the analyzed loop and annotated with a DepTouch directive, is symbolically executed the access pointer of the statement, `AccPointer`, points to a node, n, that has to represent a single memory location. Each node n of an S_j's RSG graph, has a **Touch Set** associated with it, $TOUCH_n$. The `DepTouch` directive is also interpreted by the analyzer leading to the updating of that $TOUCH_n$ 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.

```
fun TOUCH_Updating(TOUCH_n , AccAttS_j , DepGroup_g)
 if TOUCH_n == ∅ 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);
       /* update the access pairs group set */
    TOUCH_n = TOUCH_n ∪ {AccAttS_j}; /* append the new access attribute */
 endif;
return(AccessPairsGroup_g);


fun AccessPairsGroup_Updating(TOUCH_n , AccAttS_j , DepGroup_g)
 ∀ AccAttS_k ∈ TOUCH_n
    if AccAttS_k ∈ DepGroup_g then /* AccAttS_k and AccAttS_j ∈ DepGroup_g */
       AccessPairsGroup_g = AccessPairsGroup_g ∪ {<AccAttS_k,AccAttS_j>};
          /* A new ordered pair is appended */
    endif;
return(AccessPairsGroup_g);
```

Fig. 6. TOUCH and AccessPairsGroup updating functions.

As we note in Fig. 6, if the Touch set was originally empty we just append the new access attribute $\mathrm{AccAttS}_j$ of the DepTouch directive. However, if the Touch set does already contains other access attributes, $\{AccAttS_k\}$, two actions take place: first, an updating of the $AccessPairsGroup_g$ associated with the $DepGroup_g$ happens; secondly, the access attribute $\mathrm{AccAttS}_j$ 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. Here we check all the access attributes of the statements that have touched previously the node n. If there is any access attribute, $\mathrm{AccAttS}_k$ which belongs to the same $DepGroup_g$ that $\mathrm{AccAttS}_j$ (the current statement), then a new access pair is appended to the $AccessPairsGroup_g$. The new pair is an ordered pair $<\mathrm{AccAttS}_k, \mathrm{AccAttS}_j>$ 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, and so 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.

4. In the last step, our LCD_Test function will check each one of the $AccessPairGroup_g$ updated in step 3. This function is detailed in the code of Fig. 7. If an $AccessPairGroup_g$ is empty, the statements associated with the corresponding $DepGroup_g$ does not provoke any LCD. On the contrary, depending on the pairs comprised by the $AccessPairsGroup_g$ we can raise some of the dependence patterns provided in Fig. 7, thus LCD is reported.

```
fun LCD_Test(AccessPairGroup_g)
 if <WriteS_i,ReadS_j> ∈ AccessPairGroup_g
   then return(FlowDep); /* Flow dep. detected between S_i and S_j */
 if <ReadS_i,WriteS_j> ∈ AccessPairGroup_g
   then return(AntiDep); /* Anti dep. detected between S_i and S_j */
 if <WriteS_i,WriteS_j> ∈ AccessPairGroup_g
   then return(OutputDep); /* Output dep. detected between S_i and S_j */
 if <WriteS_i,WriteS_i> ∈ AccessPairGroup_g
   then return(OutputDep); /* Output dep. detected between S_i and S_i */
 endif
return(NoDep); /* no LCD detected */
```

Fig. 7. LCD test.

We note that the `LCD_Test` function must be performed for all the *AccessPairGroup*s updated in step 3. When we verify for all the *AccessPairGroup*s, that none of the dependence patterns is found, then our algorithm informs that the loop does not contain LCD dependences (`NoLCD`) due to heap-based pointers.

3.1 An example

Let's illustrate via a simple example how our 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` read 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 our 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 our 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 our 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 $DepGroup_{val}$={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 our 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.

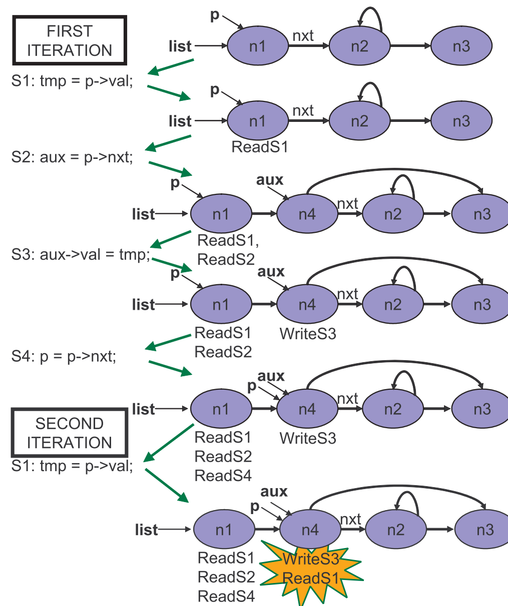

Fig. 9. Initial RSG at the loop entry and the resultant RSG graphs when executing S1, S2, S3 and S4 in the first loop iteration, and when S1 is executed in the second loop iteration.

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 our example, WriteS3 is in the $DepGroup_{val}$. In this case, since the new access attribute that is touching 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>}.

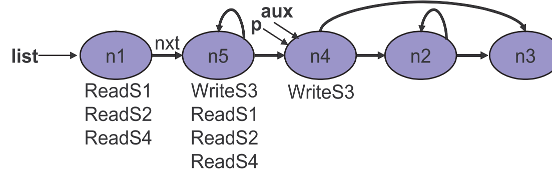

Fig. 10. Resultant RSG when the fixed point is reached.

Our algorithm applies now the fourth step: the LCD test (Fig. 7). Our LCD test reports a `FlowDep` (flow dependence), because the only access pair group, $AccessPairGroup_{val}$ in the ACCESSPAIRSGROUP set, contains a <WriteS3, ReadS1> pair. As we see, our dependences detection algorithm accurately captures the LCD that appears in the loop.

3.2 Some preliminary results

We have applied our dependences detection algorithm to some sample codes, which we show in Fig. 11. The goal of these preliminary experiments is to illustrate the accuracy of our method in the detection of data dependences in codes

that traverse (and/or modify) complex data structures. In each code, the sentences have been atomized and annotated with the corresponding directives (for simplicity, we do not display them) in step 1. The code from Fig. 11(a) traverses a DAG data structure, which is shown in Fig. 12(a) and whose RSG at the loop entry is shown in the same figure. Note that in the RSG, the dotted edges reaching $n4$ means that SHSEL(n4,ch)=true, which captures the fact that a memory location represented by $n4$ can be reached from different memory locations by following the `ch` selector. We assume that there is only one RSG graph in the RSGs set at this program point. The write statement that may induce an LCD is S3 (S1 access the same "i" field but using an access pointer of a different data type). Therefore a dependence group is created: $DepGroup_i$={<WriteS3>}. The associated $AccessPairsGroup_i$ is empty at this point. At the end of step 3, our algorithm returns $AccessPairsGroup_i$={WriteS3, WriteS3}. This is due to the fact that we can write the same `p->ch->i` location in several iterations of the loop. Applying the step 4, our LCD test function reports an output dependence (`OutputDep`) for such access pair group.

```
p = list;
while (p->nxt != NULL)
{
S1: tmp = p->i;
S2: aux1 = p->ch;
S3: aux1->i = tmp;
S4: p = p->nxt;
}
```

(a)

```
p = list;
while (p != NULL)
{
S1: tmp = p->val;
    if (p->prv != NULL){
S2:   aux2 = p->prv;
S3:   aux3 = aux2->ch;
S4:   aux3->i = tmp;}
S5:  p = p->nxt;
}
```

(b)

```
p = list;
while (p->nxt != NULL)
{
S1: tmp = p->num;
    if (cond){
S2:   aux4 = p->nxt;
S3:   aux4->num = tmp;}
    else{
S4:   aux5 = p->prv;
S5:   aux5->num = tmp;}
S6: p = p->nxt;
}
```

(c)

Fig. 11. (a) Traversal of a DAG data structure; (b) Cyclic access in a Cyclic data structure; (c) Conditional cyclic access in a Cyclic data structure.

Another case is illustrated in the code of Fig. 11(b). The cyclic data structure and the corresponding RSG at the loop entry are shown in Fig. 12(b). Although this RSG is similar to the one of the previous example, now SHSEL(n4,ch)=false, accurately capturing the new topology of this data structure. In the step 2 of our algorithm, a dependence group $DepGroup_i$={WriteS4} is created and the associated $AccessPairsGroup_i$ is initially set to the empty set. At the end of step 3, our algorithm reports that $AccessPairsGroup_{val}$ is still empty because our shape analysis does not touch twice the same node with the WriteS4 attribute. Thus now, the LCD test function in step 4 reports a no dependence (`NoDep`). As there is not another access pair group, our test informs that there is not LCDs due to heap-based pointers (`NoLCD`).

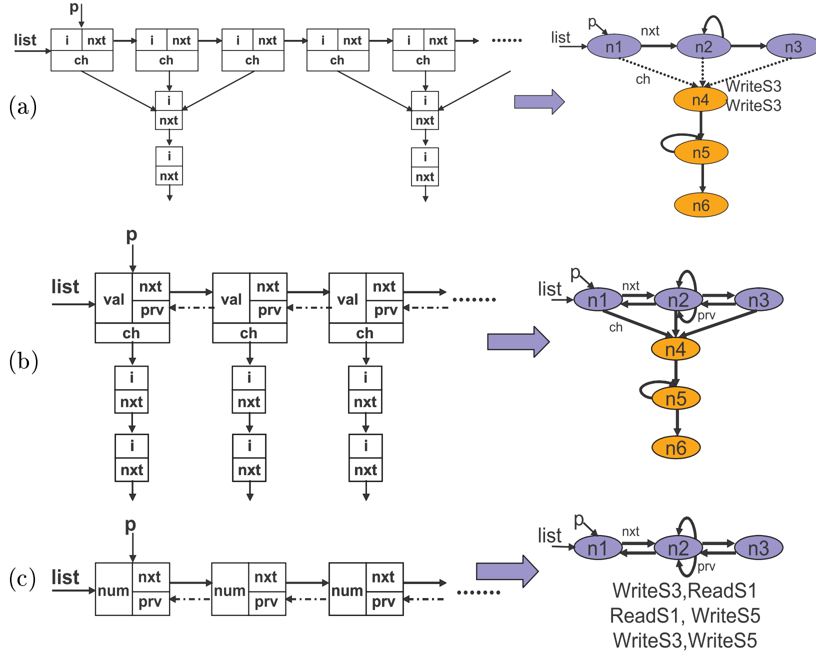

Fig. 12. Data structures and the corresponding RSG: (a) DAG; (b) Cyclic; (c) Cyclic.

Finally, a new case is illustrated in the code of Fig. 11(c). The data structure and the corresponding RSG at the loop entry are shown in Fig. 12(c). In this case, the statement S1 read a field memory location (given by *num*) that may be written conditionally by statements S3 or S5. Thus some dependence may arise. Now, some traversal of the data structure may contain a cycle (due to the (*nxt*, *prv*) selectors). In step 2 the dependence group is defined as $DepGroup_{num}$={ReadS1, WriteS3, WriteS5}. At the end of step 3, our algorithm reports that the pairs <WriteS3, ReadS1>, <ReadS1, WriteS5> and <WriteS3, WriteS5> are in the $AccessPairsGroup_{num}$. Thus, our LCD test function accurately detects a flow dependence (`FlowDep`), an anti-dependence (`AntiDep`) and an output dependence (`OutDep`).

4 Related works

Some of the previous works on dependences detection on pointer-based codes, combine dependence analysis techniques with pointer analysis [4], [5], [9], [6], [3], [7]. Horwitz et al. [5] developed an algorithm to determine dependences by detecting interferences in reaching stores. Larus and Hilfinger [9] propose to identify access conflicts on alias graphs using path expressions to name locations. Hendren and Nicolau [4] 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 [3], [7], [8] have proposed dependence analysis tests 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 [3] 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 test 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. For instance, even asserting as acyclic the loop example from Fig.11(b), this method would have detected a LCD in the code (due to the `prv` and `ch` pointer fields in the access path of statement `S4`), while our method can successfully proves that there is no dependences. Another limitation of this approach is that data structures must remain static during the data traversal inside the analyzed loops.

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 [7], [8]. 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. For instance, in the code example from Fig.11(b), this method would detect a LCD due to a cycle in the traversal pattern (due to the `prv` and `nxt` selectors). On the other hand, their analysis can overestimate the shape of the traverse when the data structure is modified along the traverse, and in these situations, the shape algorithm detect DAG or Cycle traversal patterns, in which case dependence is reported.

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 modify generic heap-based dynamic data structures. Our algorithm is able to identify accurately the dependences that appears even in loops that navigate (and modify) cyclic structures in traversals that contain cycles, as we have seen in the code examples from Fig. 11. Besides we can successfully discriminate among flow, anti and output dependences.

5 Conclusions and Future Works

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 modify heap-based dynamic data structures. Our algorithm is able to identify precisely dependences even in loops that navigate (and modify) 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 a preliminary implementation of our compilation algorithms and we have checked the success in the LCDs detection in several synthetic small codes. We are planning to conduct a large set of experiments based on C benchmarks, in order to demonstrate the effectiveness of our method in real applications.

References

1. 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.
2. 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.
3. 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.
4. L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1:35–47, January 1990.
5. 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.
6. 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.
7. 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.
8. Y. S. Hwang and J. Saltz. Identifying parallelism in programs with cyclic graphs. *Journal of Parallel and Distributed Computing*, 63(3):337–355, 2003.
9. 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.
10. 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.
11. 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.
12. 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.