A Novel Approach for Detecting Heap-based Loop-carried Dependences[*]

A. Tineo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata
Dpt. of Computer Architecture, University of Málaga,
Complejo Tecnologico, Campus de Teatinos, E-29071. Málaga, Spain.
{tineo,corbera,angeles,asenjo,ezapata}@ac.uma.es

Abstract

The problem of data dependences in pointer-based codes 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. 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. 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 dependence test which we describe in this paper. We also discuss its application to three different programs: the sparse matrix-vector product, mst *from Olden and* twolf *from the SPEC CPU2000 suite.*

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. 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. The problem of calculating pointer-induced aliases, called pointer analysis, has received significant attention over the past few years [11], [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].

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], [6], 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 abstract interpretation of the program statements over the graphs that represent the data structure at each program point. In other words, our ap-

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

proach 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 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 describe in this paper. In addition, we discuss the behavior and effectiveness of our test when applied to some sample programs. For these experiments we considered the sparse matrix by vector product and benchmark programs like *mst* from the Olden suite [1] and *twolf* from the SPEC CPU2000 suite [9]. In the light of these experiments and in the context of real applications, we believe that our approach provides more accurate results when compared to previous techniques while the analysis times are still reasonable.

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, and to discuss their applicability. 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. In Section 5 we discuss the application of our test to some realistic programs. 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 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->field = NULL`, `x = y`, `x->field = y` and `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 [10] in order to automatically normalize and preprocess 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.

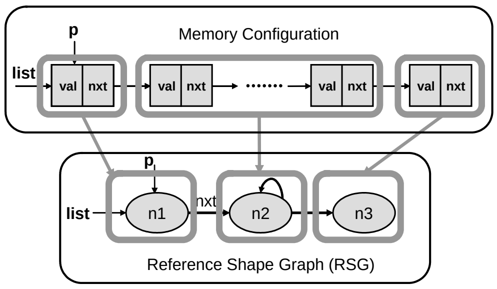

Figure 1. Working example data structure and the corresponding RSG.

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].

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 statements of the program over the graphs is carried out. In fact, 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 execution of the statement must transform the graphs [2]. This 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 process is 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.

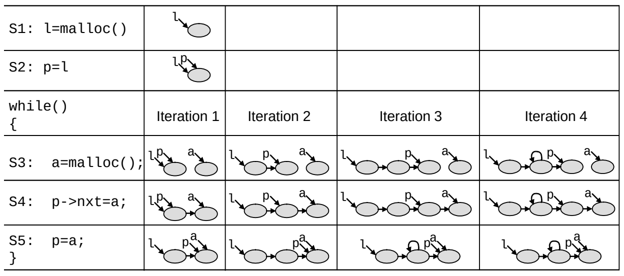

Figure 2. Building an RSG for each statement

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.

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. 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: a) all the access attributes belong to Dependence Touchs with the same access field (g) and with access pointers of the same data type; and b) 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 access 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 would 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.

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 [2] and briefly introduced in Section 2. In this step, our algorithm calls 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 the shape analysis presented in [2] in order to deal with the dependence analysis.

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). When a statement S_j, belonging to the analyzed loop and annotated with a DepTouch directive, is symbolically executed, then 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

```
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
```

Figure 3. Our dependences detection algorithm.

```
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 $j \neq i$ /
    (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);
```

Figure 4. `Create_Dependence_Groups` function.

Group, $DepGroup_g$. Let n be the 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. 5 takes place.

As we note in Fig. 5, if the $TOUCH_n$ set was originally empty we just append the new access attribute AccAttS$_j$ of the `DepTouch` directive. However, if the $TOUCH_n$ 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 AccAttS$_j$ is appended to the $TOUCH_n$ set of the node, i.e., $TOUCH_n = TOUCH_n \cup \{AccAttS_j\}$.

The algorithm for updating the $AccessPairsGroup_g$ is shown in Fig. 5. Here 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), 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, 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. 6. 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. 6, thus LCD is reported.

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. 7(a) represents a loop that traverses the data structure of Fig. 1. This is, this loop is going to be exe-

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

fun TOUCH_Updating(TOUCH$_n$, AccAtt$S_j$, $DepGroup_g$)
 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$_n$, AccAtt$S_j$, $DepGroup_g$);
       /* 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$_n$, AccAtt$S_j$, $DepGroup_g$)
 ∀ AccAtt$S_k$ ∈ TOUCH$_n$
    if AccAtt$S_k$ ∈ $DepGroup_g$ then /* AccAtt$S_k$ and AccAtt$S_j$ ∈ $DepGroup_g$ */
       $AccessPairsGroup_g$ = $AccessPairsGroup_g$ ∪ {<AccAtt$S_k$,AccAtt$S_j$>};
          /* A new ordered pair is appended */
    endif;
return($AccessPairsGroup_g$);
```

Figure 5. Shape_Analysis **function extension,** TOUCH_Updating **and** AccessPairsGroup_Updating **functions.**

```
fun LCD_Test(AccessPairsGroup$_g$)
 if <Write$S_i$,Read$S_j$> ∈ AccessPairsGroup$_g$
   then return(FlowDep); /* Flow dep.  */
 if <Read$S_i$,Write$S_j$> ∈ AccessPairsGroup$_g$
   then return(AntiDep); /* Anti dep.  */
 if <Write$S_i$,Write$S_j$> ∈ AccessPairsGroup$_g$
   then return(OutputDep); /* Output dep.  */
 if <Write$S_i$,Write$S_i$> ∈ AccessPairsGroup$_g$
   then return(OutputDep); /* Output dep.  */
 endif
return(NoDep); /* no LCD detected */
```

Figure 6. LCD test.

cuted 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.

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. 7(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 sim-

```
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)

Figure 7. (a) Loop traversal of a dynamic data structure; (b) Instrumented code.

ple statement in the loop that accesses the heap, as we can also appreciate in Fig. 7(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 each statement will be updated at each symbolic execution and the loop analysis will finish when all the graphs in all the RSGs do not change any more.

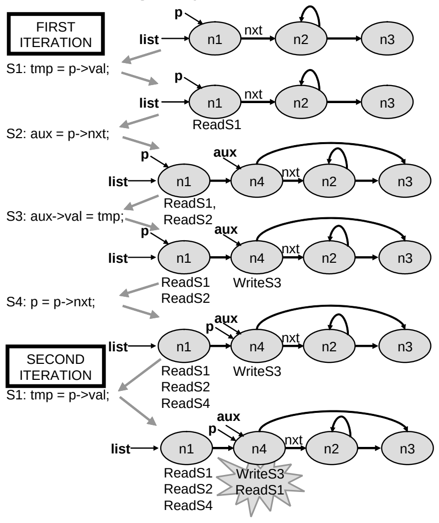

Figure 8. Initial RSG at loop entry and resultant RSG graphs when executing statements.

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. 8. 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. 9. We also get at the end of the analysis that $AccessPairsGroup_{val}$= {<WriteS3, ReadS1>}.

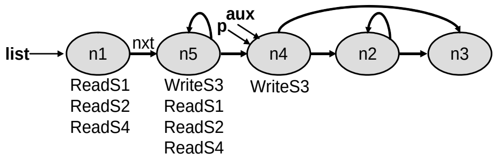

Figure 9. Resultant RSG when the fixed point is reached. The TOUCH sets are illustrated.

Our algorithm applies now the fourth step: the LCD test (Fig. 6). 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.

4 Related work

Some of the previous works on dependences detection on pointer-based codes, combine dependence analysis techniques with pointer analysis [4], [7], [5]. 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 goal in our approach.

More recently, some authors [3], [6] 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. On the other hand, Hwang and Saltz proposed a new technique to identify LCDs in programs that traverse cyclic data structures [6]. 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 (again Tree, DAG or Cycle). Once they have extracted the traversal-pattern shape information, dependence analysis is applied to detect LCDs. One limitation of these approaches is that they are just able to analyze loops that navigate data structures in a "clean" tree-like traverse.

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) complex structures in traversals that contain cycles, as we have demonstrated in [8]. Besides we can successfully discriminate among flow, anti and output dependences which is vital in optimizing and parallelizing compilers. The previous works were unable to compute that information. Our goal here is to put our analysis to work with larger codes and to study the applicability of our method to real programs.

5 Experimental results

We have implemented all the algorithms presented in this paper. We have tested our prototype implementation on several small examples (see [8]). In order to prove the effectiveness of our method on larger C programs, we have considered 3 codes. The first one is a custom-made program that represents the kernel of typical real world applications which deal with dynamic data structures: the sparse matrix-vector product. It is available through our website[1]. The last two programs were taken from well established benchmarks, *mst* from the Olden suite [1] and *twolf* from the SPEC CPU2000 suite [9].

For these tests, we focused our analysis on certain loops that carry a significant amount of execution time. That information was gathered through profiling. At a first stage, the programs were digested by our preprocessing tool based on ANTLR to discard statements not involved in heap accesses and to mark the useful statements with `DepTouch` information. At this point of development, interprocedural analysis is not yet supported so, as a temporal solution, inlining of functions was performed. Besides, some manual adjustments were needed in order to fully adapt the input code to the test dependence module. However, it should be noted that the method itself is fully automatic. Table 1 shows some statistics for the tests, run in a Pentium IV 2.8GHz with 256MBytes. The measured **Times** include the underlaying shape analysis. In addition, the times take into account the creation of the data structures which are accessed in the analyzed loops. In fact, **No. Stmt.** represents the number of simple statements of the analyzed input code, including the statements of the studied loop as well as the statements for the corresponding data structures creation. **No. It.** represents the number of iterations that our method needs to reach a fixed point. **No. Graphs** is the number of RSG graphs generated for each code, including the temporal graphs that may appear during a statement execution, whereas **Graphs/Stmt.** represents the RSG graphs per statement ratio.

Our first program computes the product of a sparse matrix by a sparse vector. We apply our analysis to the loop that computes the product. In this loop, the elements of the output data structure are created while the matrix and the input vector are traversed. For the studied loop, the analysis determines that there is no dependence because the nodes that are read (those from the matrix and the input vector) are different in every iteration from the nodes that are written (those from the output vector). Thus, the method reports `NoDep`, therefore the loop could be parallelized. As we see in Table 1, the analysis time is reasonable even though the number of RSG graphs generated during the analysis, and the number of iterations to find the fixed point are high.

Our next code, *mst* is a program from the Olden benchmark suite. In *mst*'s data structure there is an array of vertexes. Each of them holds a hash table that must store information about every other vertex. The profiling of the code discovered that more than 85% of the execution time was spent in an inner loop of the `AddEdges` function. So we select that loop for the study with our tool. In that loop, hash tables for the nodes are populated with information about every other node. It is undecidable at compile-time the order of writing for the entries in the hash tables, so all kind of dependences arise (flow, anti, output). The method reports all these dependences and the statements for which each dependence appears. This latter information is very important to improve data-cache performance. We see in Table 1 that this is the code for which the number of generated RSG graphs and the number of iterations to reach the fixed point are the smallest. As a consequence, the analysis time is significantly short.

The last code considered is *twolf*, from SPEC CPU2000. It is considerably larger in size than the previous programs (roughly 20,000 lines of code). Profiling was performed to find the most computationally expensive loops. We found that the loop contained in the `new_dbox_a` function consumed more than 30% of the execution time, so we focused

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

Program	Time	No. Stmt.	No. It.	No. Graphs	Graphs/Stmt.
`matrix-vector`	1 min 47 sec	120	315	170450	1420
`mst`	0.7 sec	62	70	3314	53
`twolf`	5 min 54 sec	154	149	112803	732

Table 1. Time and other measures for the codes.

our analysis in that loop. On the other hand, *twolf*'s data structure is more complicated than those from the previous examples. For the studied loop, 3 dynamic interconnected structures are involved. A list is traversed reading values to index another list. The elements of the indexed list are then processed to compute some values. It is possible to access the same elements in different iterations of the list, so dependences arise again and the method reports all of them accurately. The results for this code (Table 1) are very telling. It is the biggest code, but it is not the one with the highest number of generated RSG graphs. In fact, the ratio of RSGs per statement is smaller than for the matrix-vector code. As a consequence, the number of iterations to reach the fixed point is smaller than in the matrix-vector case. However, the analysis time is the worst. The reason is due to the complex nature of the `twolf`'s RSG graphs. We have verified that the majority of the nodes in the `twolf`'s graphs, are heavily connected with each other, what produces an important time consumption every time that internal operations of the shape analysis take place (materialization of nodes, summarization of nodes, comparison of graphs, etc.). This observation tell us that optimization of these internal operations of the shape analysis tool must be addressed.

Summarizing, these results have shown us that a smaller number of graphs per statement will need less iterations to reach the fixed point. But this is not a guarantee that the analysis times will be smaller. Another surprising result is that although the number of generated graphs is very high, we think that the times are still reasonable. Let's keep in mind that our approach is able to provide very accurate data dependence information in the context of real codes which traverse and modify generic and complex heap-based data structures. One key aspect of our method is that it allows elements of the data structure to be created inside the traversals (like in the matrix-vector product), as well as conveniently distinguish between flow, anti and output dependences, which is basic for doing data-cache optimizations. Let's recall that all these aspects are left out in every other approach we know. 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.

6 Conclusion and Future Work

We have presented a compilation technique that is able to identify LCDs in loops that traverse and modify general pointer-based dynamic data structures. 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 conducted some tests that prove it can be useful for real-life programs. However, more work is necessary in order to optimize the internal operations of the shape analyzer to achieve a faster test.

References

[1] M. C. Carlisle and A. 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. 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, 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] 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.

[5] 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.

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

[7] 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.

[8] A. Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plata, and E. 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.

[9] S. P. E. C. (SPEC). *SPEC CPU2000 V1.2 Documentation*, 2000. http://www.spec.org/cpu2000/docs/.

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

[11] 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.