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

A. Navarro, F. Corbera, R. Asenjo, 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,oscar,ezapata}@ac.uma.es

Abstract. The approach presented in this paper focus on detecting parallelism on loops that traverse pointer-based 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. Until now, the majority of parallelism detection techniques based on shape analysis have used as shape information a coarse characterization of the data structured being traversed (Tree, DAG, Cycle). This causes a loss of accuracy that prevents the parallelization of loops, specially when the data structure being visited is not a "clean" tree or may contain cycles. Our approach, on the contrary, 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. Moreover, the novelty is that our approach executes, at compile time, the statements of the loop being analyzed, and let us to annotate the real 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 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 [11], [10], [2]. Pointer analysis can be divided into two distinct subproblems: stack-directed analysis and heap-directed analysis. We

* This work was supported in part by the European Union under contract 1FD97-2103, and by the Ministry of Education of Spain under contract TIC2000-1658.

focus our research in the later, 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 try to estimate 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 parallelism on loops that traverse pointer-based 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. For instance, if a pointer `p` points to a tree-like data structure, then any two accesses of the form `p->f` and `p->g` will always led to disjoint sub-pieces of the tree (assuming `f` and `g` are distinct fields or selectors). Or if the pointer `p` points to a generic data structure, then two distinct field accesses `p->f->f` and `p->g` can lead to a common node. However, if the data structure does not contain cycles and it is traversed using a sequence of links: `p->f`, `p->f->f`, ..., then every subsequence visits a distinct node. As we see, the techniques based on some kind of shape information may be useful to disambiguate heap accesses in different iterations of a loop, hence to provide that there are not data dependences between iterations and that the loop can be executed in parallel.

Until now, the majority of parallelism detection techniques based on shape analysis [3], [8], use as shape information a coarse characterization of the data structured being traversed (Tree, DAG, Cycle). One advantage of this type of analysis is that enables faster data flow merge operations and reduces the storage requirements for the analysis. However, it also causes a loss of accuracy that prevents the parallelization of loops, specially when the data structure being visited is not a "clean" tree or may contain cycles.

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 dependences. The novelty is that our approach executes, at compile time, the statements of the loop being analyzed, and let us to annotate the real 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, by the first time, in this paper.

Summarizing, the goal of this paper is to present our compilation algorithms to detect parallelism 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 II briefly describes the key ideas under our shape analysis framework. With this background, in section III we present our compiler techniques to automatically identify the parallel loops in codes based on dynamic data structures. Finally, in section V 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 recursive data structures that are connected through pointers defined in languages like C or C++. We assume that our programs have been normalized in such a way that each statement dealing with pointers contains only simple access paths. Therefore, 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. More complex pointer instructions can be built upon these simple ones and temporal variables

Basically, our method is based on approximating by graphs all possible memory configurations that can appear after the execution of a statement in the code. We call a collection of dynamic structures a *memory configuration*. These structures comprise several memory chunks, that we call *memory locations*, which are linked by references. Inside these memory locations there is room for data and for pointers to other memory locations.

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

2.1 RSGs and node properties

The RSGs are graphs in which nodes represent memory locations which have similar reference patterns. To determine whether or not two memory locations should be represented by a single node, each one is annotated with a set of properties. Now, if several memory locations share the same properties, then all of them will be represented by the same node. This way, a possibly unlimited memory configuration can be represented by a limited size RSG, because the number of different nodes is limited by the number of properties of each node.

These properties are related to the “reference pattern” used to access the memory locations represented by the node. Hence the name *Reference Shape Graph*. The most important property regarding our parallelism detection algorithms is the TOUCH property which let to annotate the nodes of the data structure which are written and/or read by the pointer statements inside loops. A more detailed description of all properties can be found in [1].

As we have said, all possible memory configurations which may arise after the execution of a statement are approximated by a set of RSGs. We call this set *Reduced Set of Reference Shape Graphs* (RSRSG), since not all the different RSGs arising in each statement will be kept. On the contrary, several RSGs related to different memory configurations will be fused when they represent memory locations with similar reference patterns.

Let us illustrate all this with an example. In Figure 1 we can see a simple code with seven pointer statements. Our analyzer symbolically executes each statement to build the RSRSG associated with them. Actually, after the execution of the third statement we obtain an RSRSG with a single RSG which represents three different memory locations by three nodes; all of them of the same type, with the same *nxt* selector, but pointed to by different pointer variables. Now, this RSRSG is modified in three different ways because there are three different paths in the control flow graph, each one with a different pointer statement. All these paths join in statement 7, and after the execution of this statement we obtain an RSRSG with two RSGs. This is because the RSGs coming from statements 5 and 6 are compatible and can be summarized into a single one.

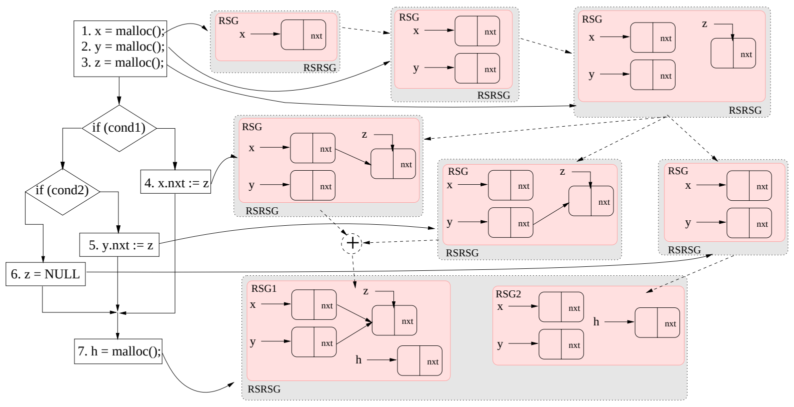

Fig. 1. Building an RSRSG for each statement of an example code.

2.2 Generating the RSRSGs: the symbolic execution

To move from the “memory domain” to the “graph domain”, the calculation of the RSRSGs associated with a statement is carried out by the **symbolic**

execution of the program over the graphs. In this way, each program statement transforms the graphs to reflect the changes in memory configurations derived from statement execution. The **abstract semantic** of each statement states how the analysis of this statement must transform the graphs. Due to space constraints we cannot formally describe the abstract semantics of each one of the statements. The details can be found in [1].

The abstract interpretation is carried out iteratively for each statement until we reach a fixed point in which the resulting RSG's associated with the statement does not change any more.

3 Parallelism Detection

As we have mentioned, we focus on detecting parallelism on loops that traverse heap-based recursive data structures. In general, for finding loop parallelism we need to detect the presence of *loop-carried dependences* (henceforth referred as LCDs). Two statements in a loop have 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. 2. 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) and the RSRSG at the loop entry, are the input to this algorithm.

```
fun Parallelism_Detection (SIMPLESTMT, RSRSG)
1. $\forall$ $S_i$ $\in$ SIMPLESTMT that accesses the heap
        Attach($S_i$, DepTouch(AccPointer,AccAttS$_i$,AccField));
2. DEPGROUP = Create_Dependence_Groups(DEPTOUCH);
   $\forall$ $DepGroup_g$ $\in$ DEPGROUP
        $AccessPairsGroup_g$ = $\emptyset$ ;
3. ACCESSPAIRSGROUP = Shape_Analysis(RSRSG,SIMPLESTMT, DEPTOUCH, DEPGROUP);
4. $\forall$ $AccessPairsGroup_g$ $\in$ ACCESSPAIRSGROUP
        Dep$_g$ = LCD_Test($AccessPairsGroup_g$);
   if $\forall$ g, Dep$_g$ == NoDep then
        return(ParallelLoop);
   else
        return(Dep);
   endif;
end
```

Fig. 2. Our parallelism 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, the `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, ReadS4, 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, 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 and one of the accesses is a write. We outline in Fig. 3 the function `Create_Dependence_Groups`. It creates Dependence Groups, using as a input the set of Dependence Touch directives, DEPTOUCH. Note that 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. In general, two Dependence Groups, $DepGroup_g$ and $DepGroup_f$ (being $g \neq f$) could lead to different LCDs by accessing each group to different fields of the traversed data structures. As we see in Fig. 3 the output of the function is the set of all the Dependence Groups, named DEPGROUP.
   Associated to each $DepGroup_g$, our algorithm initializes a set called $AccessPairsGroup_g$ (see Fig. 2). 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 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

```
fun Create_Dependence_Groups(DEPTOUCH)
 ∀ DepTouch(Accpointer$_i$,AccAttS$_i$,AccField$_i$)
    if [(AccAttS$_i$ == WriteS$_i$) or
    ∃ DepTouch(Accpointer$_j$,AccAttS$_j$,AccField$_j$) being $j \neq i$ /
    (AccField$_i$ == AccFieldS$_j$) and (AccAttS$_i$ == WriteS$_i$ or AccAttS$_j$ == WriteS$_j$)] then
       $g$ = AccField$_i$;
       if $\nexists$ $DepGroup_g$ then
          $DepGroup_g$ = {AccAttS$_i$};
       else
          $DepGroup_g$ = $DepGroup_g$ ∪ {AccAttS$_i$};
       endif;
    endif;
return(DEPGROUP);
```

Fig. 3. Create_Dependence_Groups function.

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. With this and with the DepTouch directive, the task of the analyzer is to symbolically execute each statement updating the graphs and at the same time, 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 graphs RSRSG, 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. We outline that function in Fig. 4.
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 RSRSG$_{in}$ from the previous statement and modifies it (producing a RSRSG$_{out}$). When a statement S_j is symbolically executed the access pointer of the statement, AccPointer, points to a node, n, that has to represent a single memory location (thanks to the materialization operation). Each node n of a graph in the RSRSG$_{out}$, has a **Touch Set** associated with it, $TOUCH_n$. If S_j has attached a DepTouch directive, it 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 assume that AccAttS$_j$ belongs to a Depen-

```
fun Shape_Analysis(RSRSG,SIMPLESTMT, DEPTOUCH, DEPGROUP)
 RSRSG$_{in_1}$ = RSRSG;
 $\forall$ $S_j$ $\in$ SIMPLESTMT
    RSRSG$_{out_j}$ = Execute($S_j$, RSRSG$_{in_j}$);
    if DepTouch(Accpointer,AccAttS$_j$,AccField) attached to $S_j$ then
       $AccessPairsGroup_g$ = TOUCH_Updating(TOUCH$_n$, AccAttS$_j$, $DepGroup_g$);
    endif;
    RSRSG$_{in_{j+1}}$ = RSRSG$_{out_j}$;
return(ACCESSPAIRSGROUP);
```

Fig. 4. The Shape_Analysis function.

dence 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 $TOUCH_n = \{AccAttS_k\}$ the set of access attributes with belong to the $TOUCH_n$ set, where k represent 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 set was originally empty we just append the new access attribute 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 AccAttS$_j$ is appended to the Touch set of the node, $TOUCH_n = TOUCH_n \cup \{AccAttS_j\}$. The algorithm for the updating of 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 to the same dependence group, 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 $AccessPairGroup_g$ we can raise some of the dependence patterns provides in Fig. 6, thus LCD is reported.

```
fun TOUCH_Updating(TOUCH$_n$, AccAttS$_j$, $DepGroup_g$)
 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$_j$, $DepGroup_g$);
       /* 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$_j$, $DepGroup_g$)
 $\forall$ 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. 5. TOUCH and AccessPairsGroup updating functions.

We note that the `LCD_Test` function must be performed for all the *AccessPairGroups* updated in step 3. When we verify for all the *AccessPairGroups*, that none of the dependence patterns is found, then the loop does not contain LCD dependences and can be parallelized.

3.1 An example

Let's illustrate via an example how our approach works. Fig. 7(a) represents a loop that traverses the data structure of Fig. 8(a), whose RSRSG graph at the loop entry is shown in Fig. 8(b). That is the RSRSG that results from a previous code section in which the data structure has been created. The sentences that may provoke a conflict: `tmp = p->val` and `p->prv->g->val = tmp` access to different memory locations on each iteration, therefore there is not LCDs and the loop can be parallelized.

We first atomize 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->prv->g->val = tmp;` has been decomposed into three simple statements S2, S3 and S4. After this step, the SIMPLESTMT set will comprise five 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 code that acesses to the heap, as we can also appreciate in Fig. 7(b). For example, the statement `S2: aux2=p->prv` has been annotated with the `DepTouch(p, ReadS2, prv)`, stating that the access pointer is `p`, the access attribute is `ReadS2` (which means that

```
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,Write_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. 6. LCD test.

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

(a)

```
p = list;
while (p != NULL)
{
S1: tmp = p->val; DepTouch(p, ReadS1, val);
      if (p->prv != NULL) then
       {
S2:    aux2 = p->prv; DepTouch(p, ReadS2, prv);
S3:    aux3 = aux2->g; DepTouch(aux2, ReadS3, g);
S4:    aux3->val = tmp; DepTouch(aux3, WriteS4, val);
       }
S5:  p = p->nxt; DepTouch(p, ReadS5, nxt);
}
```

(b)

Fig. 7. (a) Cyclic Traversal of a Cycle data structure; (b) Instrumented code used to feed our shape analyzer.

the S2 statement makes a read access to the heap) and finally, that the read access field is prv.

Next we move on to the second step in which we point out that statements S1 and S4 in our code example meet the requirements to be associated to a dependence group: both of them access the same data field (val), being S4 a write access. We will call this dependence group, $DepGroup_{val}$={ReadS1, WriteS4}. Besides, the associated $AccessPairsGroup_{val}$ set will be, at this point, empty. Therefore, after this step, the DEPGROUP = $\{DepGroup_{val}\}$ and ACCESS-PAIRSGROUP = $\{AccessPairsGroup_{val}\}$.

Let's see now how the step 3 of our algorithm proceed. As we have mentioned, Fig. 9(a) represents the RSRSG that the shape analysis tool produces at this point and that precisely captures the data structure before the symbolic execution of the loop. It will be the RSRSG_{in_1} in our algorithm.

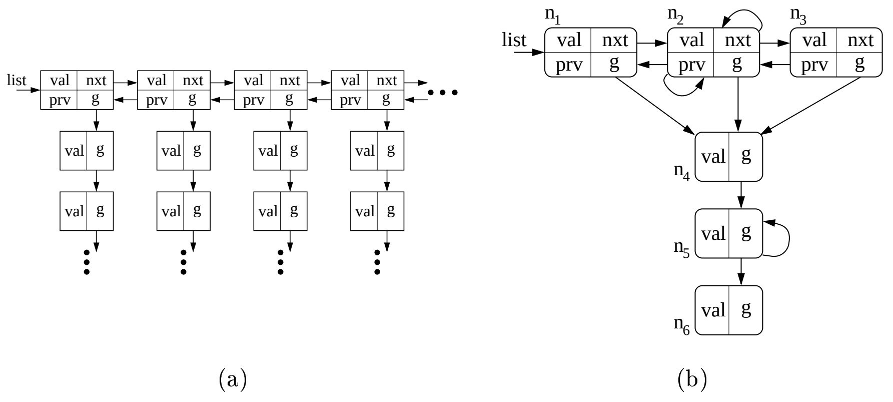

Fig. 8. (a) Working example data structure and (b) the corresponding RSRSG.

Remind that our analyzer is going to symbolically execute each of the statements of the loop iteratively until a fixed point is reached. This is, the RSGs associated with the statements will be updated at each symbolic execution and the loop analysis finish when the RSGs do not change any more.

Now, in the first loop iteration, the statement S1 is executed by the shape analyzer (note that in the first loop iteration, statements S2, S3 and S4 are not executed). An output RSG graph when this statement is symbolically executed, taking into account the attached `DepTouch` directives, is shown in Fig. 9(b). The node pointed to by `p` (n_1) is touched by `ReadS1`. When executing S5, the node pointed to by `p` will be touched by `ReadS5`, and the symbolic execution of statement `p = p->nxt` will produce the materialization of a new node (the node n_7 in Fig. 10(a)).

In the next loop iteration, the statements S1, S2, S3, and S4 are executed over the RSG graph that results from the previous symbolic execution of S5. An output resultant RSG is shown now in Fig. 10(a). In this figure we can note that the nodes pointed to by `p` (node n_7), and by `aux3` (node n_8), has been materialized. And now node n_1 is pointed to by `aux2`. The node n_7 is touched by `ReadS1` and `ReadS2`, n_1 by `ReadS3` and n_8 by `WriteS4`. When touching these nodes there are no new access pairs appended to the $AccessPairsGroup_{val}$ due to the nodes are not touched both by ReadS1 and WriteS4 (which are the access attributes that belong to the same $DepGroup_{val}$). Again, when executing S5, the node n_7 will be touched by `ReadS5` and a new node will be materialized (the node n_9 in Fig. 10(b)).

In the third loop iteration, again the statements S1, S2, S3, and S4 are executed. An output RSG is shown now in Fig. 10(b). There has been materialized the nodes pointed to by `p` (node n_9) and by `aux3` (node n_{10}). Now, `aux2` points to node n_7. The node n_9 is touched by `ReadS1` and `ReadS2`, n_7 by `ReadS3` and n_{10} by `WriteS4`. Still there is no any generation of access pairs. Next, the execution

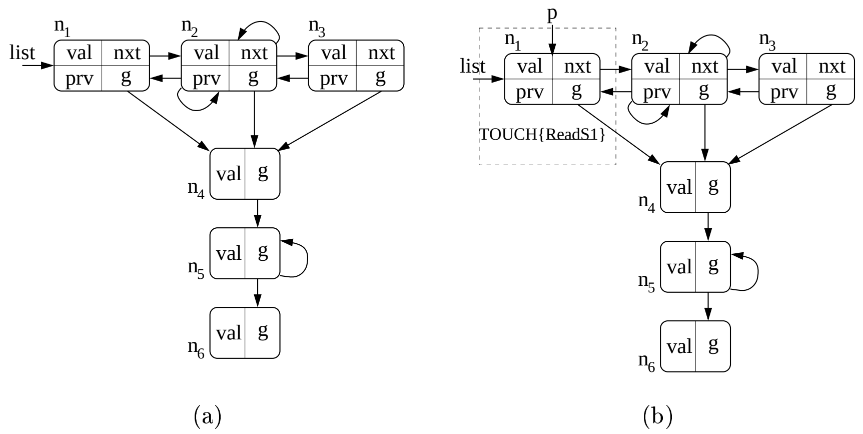

Fig. 9. (a) Initial RSRSG, at the loop entry;(b) A resultant RSG when statement S1 is executed in the first loop iteration.

of S5 will touch the node n_9 with `ReadS5` and will produce the materialization of a new node.

There is not essential differences between the third and the forth loop iterations. Let's see what happens when the instrumented statements S1, S2, S3 and S4 are executed for the fifth iteration. An output RSG is shown now in Fig. 11. The node n_{15} is representing the summarization of the already visited nodes n_7 and n_9 from Fig 10(b). The node pointed to by `p` (node n_{14}) has been materialized. Now, `aux2` points to node n_{11} and `aux3` points to the materialized node n_{16}. The node n_{14} is touched by `ReadS1` and `ReadS2`, n_{11} by `ReadS3` and n_{16} by `WriteS4`. Again, there is no any generation of access pairs.

For the next loop iteration, the shape analyzer reaches a fixed point (the new resultant graph is exactly the same that in the previous iteration), so the RSG graph that we shown in Fig.11 is an output RSG graph of the loop.

Let's apply now the fourth step of our algorithm: the LCD test (Fig. 6). Our dependence LCD test reports NoDep (no dependence), because the only access pair group set, $AccessPairGroup_{val}$, is empty. Therefore the loop does not contain LCD dependences. So the loop can be parallelized applying the corresponding transformation.

3.2 Related works

Some of previous works on parallelism 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 dependence 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

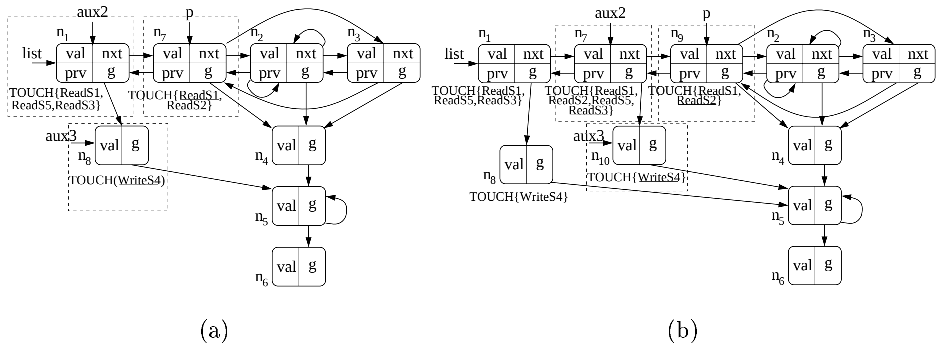

Fig. 10. A resultant RSG when statements S1, S2, S3 and S4 are executed: (a) in the second loop iteration; (b) in the third loop iteration.

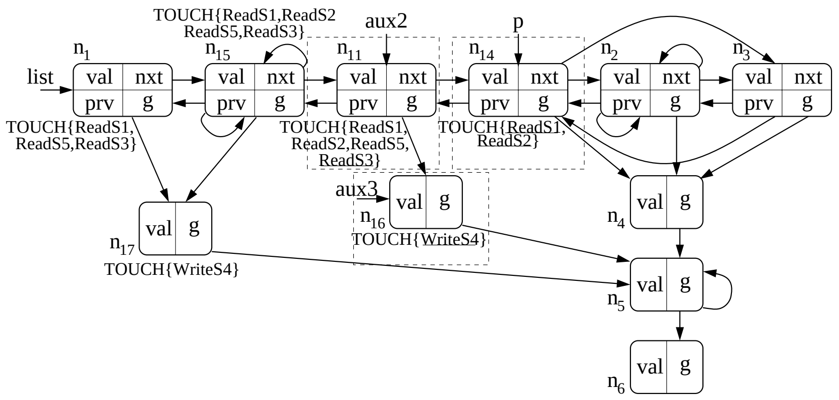

Fig. 11. A resultant RSG when statements S1, S2, S3 and S4 are executed in the fifth loop iteration. An output RSG of the loop.

and present a technique to recognize interferences between computations for programs with acyclic structures. The focus of these techniques is on identifying parallelism at the function-call level and they do not consider the detection of loop parallelism, 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 recursive data structures, and these approaches are more related to our work. The mentioned authors base their shape analysis framework in a storeless abstraction that is computed for each program point. Their shape analysis consider three possible shape attributes: Tree, DAG or Cycle. For instance, Ghiya and Hendren [3] proposed a test for identifying LCDs that relies on the shape of the data structure being traversed, as well as on the computation of the access paths for the pointers in the statements being analyzed. In short, their test identifies parallelism in programs with Tree-like data structure or loops that traverse DAG or Cycle structures and have been asserted by the programmer as acyclic. Note that the manual assertion of loops traversing cyclic data structures is a must in order to enable any automatic detection of parallelism. Another limitation 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 parallelism 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. The novelty is in that they present a technique that performs traversal-pattern sensitive shape analysis. Once they have extracted the traversal-pattern shape information, dependence analysis is applied to detect parallelism. Again, the authors consider only three possible traversal-pattern shapes: Tree, DAG and Cycle. Summarizing, their technique identifies parallelism in programs that navigate cyclic data structures in a "clean" tree-like traverse. However the 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.

For instance, in our code example of section 3.1, Ghiya and Hendren approach would have annotated the structure as Cycle, because the data structure contains a cycle (due to the (*nxt*, *prv*) selectors as we see in Fig. 8(a)). Therefore, the loop would not be further analyzed and a dependence would be reported. On the other hand, Hwang and Saltz approach, when analyzing the traversal-pattern is this loop, would have detected that the statements with the `p->prv->g` and `p->nxt` paths determine the traversal pattern, and both contain a cycle. Thus, their approach would have detected a dependence again. In both cases, the loop would have not been parallelized.

4 Conclusions and Future Works

We have presented a compilation technique that is able to identify loop-level parallelism in programs which work with general pointer-based dynamic data structures. We base our algorithms in a powerful shape analysis framework that let us to analyze quite accurately loops that traverse and modify heap-based dynamic data structures. Our algorithm is able to identify parallelism 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 to extended the scope of applicability to any program that handle any kind of dynamic data structure. Moreover, our dependence test let us to discern accurately the type of dependence: flow, anti, output.

Our current algorithms implementation can get false LCD detections when there is a dependence between two statements in the same iteration of the loop (zero-distance dependences). As a future work, we are working in a new version of our approach in which we distinguish TOUCH operations coming from different iterations from the TOUCH operations happening in the same iteration. To this end, we would use temporal access attributes which are removed when the last statement in the loop body is reached.

We have a preliminary implementation of our compilation algorithms and we have checked the success in the parallelism detection in several synthetic small codes. We are planning conducting a large set of experiments based on C dynamic data structures benchmarks, in order to provide the effectiveness of our method.

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. 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.