Compile-Time Analysis on Programs with Dynamic Pointer-Linked Data Structures

Yuan-Shin Hwang Joel Saltz

Department of Computer Science
University of Maryland
College Park, MD 20742
{shin, saltz}@cs.umd.edu

November 8, 1996

Abstract

This paper studies static analysis on programs that create and traverse dynamic pointer-linked data structures. It introduces a new type of auxiliary structures, called *link graphs*, to depict the alias information of pointers and connection relationships of dynamic pointer-linked data structures. The link graphs can be used by compilers to detect side effects, to identify the patterns of traversal, and to gather the DEF-USE information of dynamic pointer-linked data structures. The results of the above compile-time analysis are essential for parallelization and optimizations on communication and synchronization overheads. Algorithms that perform compile-time analysis on side effects and DEF-USE information using link graphs will be proposed.

Contents

1 Introduction

This paper studies static analysis on programs that create and traverse dynamic pointer-linked data structures. It proposes algorithms that perform various types of compile-time analysis to provide information, such as side effects and DEF-USE information of dynamic structures, for optimizations on sequential or data-parallel programs.

This work is motivated by the trend that more and more scientific applications use dynamic pointer-linked data structures, such as linked lists and trees. One example is the Barnes-Hut N-body solver, which stores particles in a linked list and uses a space subdivision tree (i.e. an oct-tree) to summarize the information of particles [1]. Furthermore, there have been proposals to introduce user-defined reduction functions to provide programmers the ability to formulate new combining operations on dynamic pointer-linked data structures in data-parallel languages [8, 17].

Previously proposed methods for pointer analysis, such as side effect analysis [3, 11] and conflict/interference analysis [10, 12], do not provide sufficient information for compilers to perform parallelization on sequential programs or code transformation on data-parallel programs with dynamic pointer-linked data structures. In addition to the information of side effects or conflicts on fixed-location variables at each program point, the knowledge of the process of creating and traversing dynamic linked data structures by sequences of statements is required for compile-time code optimization. Furthermore, although shape analysis can estimate the possible shapes of pointer-linked data structures [2, 9, 15, 16], it is inefficient to combine the effects of sequences of statements on dynamic pointer-linked data structures by comparing all the shape graphs of these statements.

The goal of this research is to develop a concise auxiliary structure to represent the alias information of pointers and connection relationships of the dynamic pointer-linked data structures for multiple statements, and to develop methods to identify the process of creation and traversal. Specifically, this paper is interested in finding the answers of the following questions:

- Are the pointer-linked data structures traversed by a loop static or dynamic? What are the patterns of traversal?
- For the data structures that are currently traversed by a loop, where are they created?
- Are the data structures connected or independent?

These answers can be applied to remove redundant synchronizations between loops for programs on (distributed) shared-memory multiprocessors, or to build communication schedules and embed communication calls for data-parallel programs on distributed memory systems.

The innovative idea of this paper is to differentiate associations (instances) of pointer variables, and to represent the connection relationships of these pointer associations by constructing a new form of alias graphs, called *link graphs*. In contrast to the structures proposed other researchers [2, 10, 12, 16] which build a table, a matrix, or a graph to depict alias information for every statement, only a single link graph is needed for multiple statements. Consequently, link graphs are ideal to serve as the platform for compile-time analysis on programs with dynamic pointer-linked data structures, since it usually takes multiple-statement program constructs, such as loops or recursive functions, to create or traverse pointer-linked data structures. Compilers can analyze the links of link graphs to answer the above questions, i.e. to extract the patterns of structure traversal, to determine if linked structures are static or dynamic, and to estimate the connection relationships between pointer-linked data structures. Furthermore, this paper is the first to develop a method to detect and represent the regions (i.e. sequences of statements) of programs that define or use dynamic linked data structures, and identify the DEF-USE relationships of these regions.

The link graphs are constructed from an extended form of the SSA (Static Single Assignment) intermediate representation [5]. This new SSA form extends the original SSA representation to accommodate both pointers and variables. In addition to providing information for link graph construction, this extended form of SSA can be used to perform compile-time analysis on programs with pointers. Algorithms using the original SSA form to perform code optimizations on programs without pointers, such as constant propagation and induction variable analysis [18, 19], can be modified to be applied on programs with pointers. Appendix A presents an example by adapting a constant propagation algorithm [18].

The contributions of this paper are outlined as follows:

- An auxiliary structure, called the *link graph*, is proposed to represent the connections among associations (instances) of pointer variables and fixed-location variables. It can be used to identify the process of creating and traversing dynamic pointer-linked data structures by sequences of statements. (See Section 3)

- A simple algorithm that detects side effects of programs with dynamic pointer-linked data structures is outlined. It gathers side effect information from SSA representation without concerning aliases caused by pointers, and then propagates the side effect information through the links of link graphs. (See Section 4.1)

- None of the previously developed methods provide information of how and where the currently referenced data structure is created, modified, and traversed by previously executed statements [2, 3, 10, 11, 12, 16]. This paper proposes an algorithm to identify DEF and USE regions in programs, and construct DEF-USE information of dynamic pointer-linked data structures on link graphs. This DEF-USE information is essential for compile-time optimization on synchronization and communication overheads of parallel programs with dynamic pointer-linked data structures. (See Section 4.2)

- An interprocedural compile-time analysis algorithm, safety analysis on user-defined reduction functions with dynamic pointer-linked data structures, is outlined. Without safety analysis, data-parallel language compilers have to assume all user-defined reduction functions are safe [17]. This algorithm determines if side effects or data access conflicts might cause the results of user-defined reduction functions to be unpredictable when they are executed in any order. (See Section 5.2)

Note the examples in this paper are written in Fortran 90.

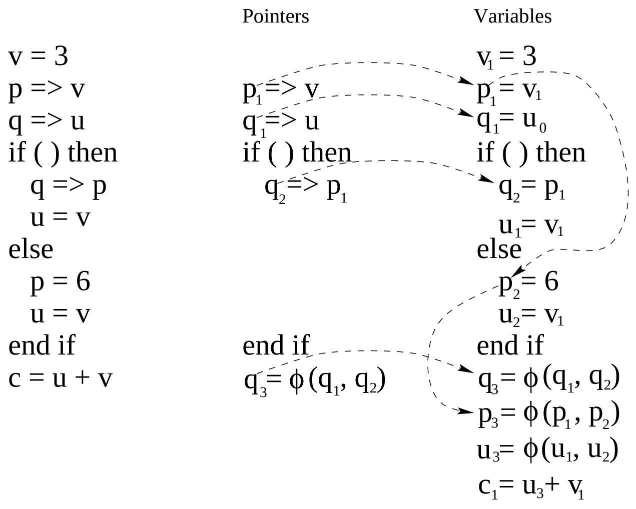
2 Extended SSA Representation

The original SSA transformation is designed specially for programs with fixed-location variables only, e.g. Fortran-77 programs, since any definitions to a variable will create a new value in one location without affecting the existing values of any other locations [5]. However, it does not apply for programs with pointers because every pointer can be dynamically assigned to point to any location. Definitions via a pointer would potentially change any existing values. Therefore, the dynamic associations of pointer variables must be resolved before converting programs into SSA form. This paper proposes an approach to represent the pointer associations by transform programs into an extended SSA format.

For programs with pointers, assignment statements can be categorized into two classes – pointer assignment statements and value assignment statements. Pointer assignment statements define the associations of pointer variables, that is, each pointer assignment redirects a pointer variable to point to a certain location in program address space. On the other hand, value assignments statements assign values to the locations corresponding to the referenced variables. Although it seems that these two classes of assignment statements work in totally different ways, an observation is that once all value assignment statements are removed from programs and all addresses of storage locations are treated as values, the pointer assignment statements work in exactly the same way as value assignment statements. Consequently, pointer assignment statements can be transformed into SSA representations just as value assignment statements.

The key is to separate pointer assignment statements from value assignment statements. For every program with pointers, the SSA representation will consist of two parts – *pointer SSA* for pointer assignment statements and *variable*

SSA for value assignments. The pointer SSA contains only the pointer assignment statements along with control flow statements, whereas the variable SSA has all statements of the program with all pointer assignment statements being converted to regular assignment statements. For example, the pointer assignment statement $p \Rightarrow v$ in the original program in Figure 1(a) has a definition $p_1 \Rightarrow v$ in pointer SSA and another one $p_1 = v_1$ in variable SSA as shown in Figure 1(b). The rationale is that, after pointer assignment two conditions hold: (1) the pointer variable p has the address of variable v, and (2) a reference to p returns the value of v. The dashed lines link the instances of pointer variables in pointer SSA to any definitions to their counterparts in variable SSA. Note that the DEF-USE information in variable SSA might not be precise because of aliases caused by pointers. However, the alias information can be collected using pointer SSA to refine the variable SSA.



(a) Program (b) SSA Representation

Figure 1: SSA Representation of A Program with Pointers

References to components of records (or structures) can be modeled as references to array elements. Any references to linked data structures through sequences of selectors, e.g. $struct\%list\%next\%node$, will be broken into sequences of statements such that each statement has at most one selector. Therefore, the only possible types of pointer assignments in pointer SSA form are

1. $ptr_1 \Rightarrow node_1$
2. $ptr_1 \Rightarrow node_1\%next$
3. $ptr_1\%next \Rightarrow node_1$
4. $allocate\ (ptr_1)$
5. $ptr_3 = \phi\ (ptr_1,\ ptr_2)$

The other type, $ptr_1\%next \Rightarrow node_1\%next$, is represented by two consecutive statements, $temp_1 \Rightarrow node_1\%next$ and $ptr_1\%next \Rightarrow temp_1$. Note that each allocation statement is modeled as a two step process: a memory location is allocated and then a pointer assignment is performed to direct the pointer to point to the location.

3 Link Graphs

This section defines the link graphs and describes the procedure to build them from the extended SSA form. Furthermore, algorithms of side effect analysis and DEF-USE information construction on programs with dynamic pointer-linked data structures will be presented in the next section as the example applications of the link graphs.

3.1 Definition

A link graph for a program or a procedure contains a set of nodes: a node for each fixed location variable (its address is a constant on pointer SSA form), a node for every definition (association) to a pointer variable, and nodes corresponding to heap allocated locations. There is a fourth type, *virtual nodes*, which will be explained later. The nodes are connected by two types of edges: an undirected ε link between two nodes means these two pointer instances are aliased (i.e. they have the same address), and a directed edge represents that sink location can be reached from source location through the selector, the name of the directed edge.

The steps to build a link graph from an extended SSA representation are as follows:

1. Construct a node for each pointer instance on SSA. Similarly, a node is created for each allocation statement.

2. For each $ptr_1 \Rightarrow node_1$ statement, create an undirected ε edge between node ptr_1 and $node_1$.

3. For each $ptr_3 = \phi\ (ptr_1,\ ptr_2)$ statement, build ε links from ptr_3 to ptr_1 and ptr_2, respectively. Furthermore, these two ε links are denoted by a ϕ mark, which means the two ε links are mutually exclusive.

4. For each $ptr_1\%next \Rightarrow node_1$ statement, build a directed link with label $next$ from ptr_1 to $node_1$.

5. For every $ptr_1 \Rightarrow node_1\%next$ statement, collect the set of nodes with an outgoing link labeled $next$ that can be reached from ptr_1 through ε edges (without passing any two ε edges denoted by a ϕ mark) and create an ε edge between ptr_1 and the sink of the $next$ link of each node in the set. However, if the set is empty, create a virtual node, build a virtual directed link with label $next$ from $node_1$ to the virtual node, and then insert an ε edge between ptr_1 and the virtual node.

For example, the link graph of the first loop of the program (in SSA format) in Figure 2(a), which is adapted from [2], can be built by following the above steps, as shown in Figure 2(b). This link graph not only clearly depicts the relationship between pointer variables w and x, but also carries important program flow control information.

Sometimes the extended SSA does not provide connection information that is required to build link graphs, i.e. when the sets in Step 5 of the above link graph construction procedure are empty. This situation usually happens when the linked data structures are constructed and then referenced in different loops or procedures. For instance, the link list constructed in the first loop (s1–s9) in Figure 2(a) is traversed in the second loop (s10–s15). The pointer variables, y and z, have no information of the linked locations on the structures, i.e. $z_2\%cdr$ and $y_1\%cdr$ are not defined within the loop. In order to convey this information, virtual nodes and directed edges are created, as the dashed circles and directed lines in Figure 2(c). Although the connection information of virtual nodes is not currently known, it can be recovered later by traversal on link graphs, e.g. the dashed undirected lines in Figure 2(c) link the virtual nodes to the allocated nodes in Figure 2(b). The main advantage of introducing virtual nodes and edges is that it gives link graphs the ability to convey the information of linked structure traversal.

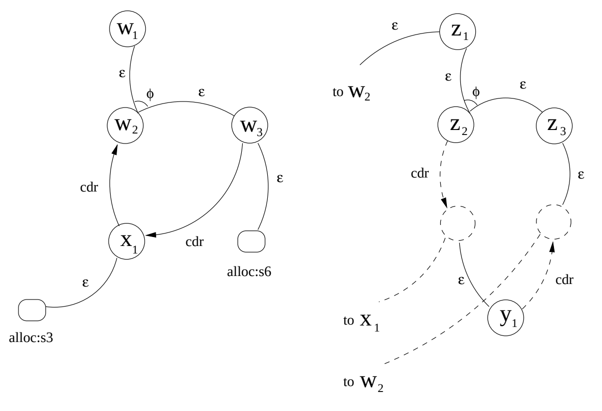
3.2 Properties

The edges of link graphs represent the associations of instances of pointer variables and the connections among locations. Furthermore, since each edge corresponds to a unique statement on pointer SSA form, the program flow

```
s1    do while (...)
s2      w2 = φ(w1, w3)
s3      allocate (x1)
s4      x1%car = 1
s5      x1%cdr => w2
s6      allocate (w3)
s7      w3%car = 2
s8      w3%cdr => x1
s9    end

s10   z1=> w2
s11   do while (...)
s12     z2= φ(z1, z3)
s13     y1=> z2%cdr
s14     z3=> y1 %cdr
s15   end
```

(a) Program (in extended SSA)



(b) Link Graph of First Loop   (c) Link Graph of Second Loop

Figure 2: SSA and Corresponding Link Graph

information is also carried by the link graphs. Consequently, important properties of dynamic pointer-linked data structures can be obtained from link graphs.

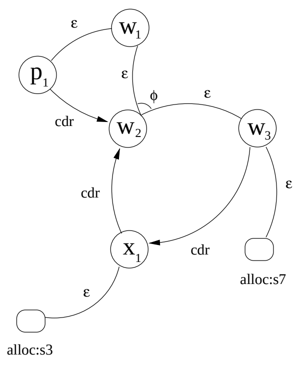
One property is that all the locations linked by the same data structure will be on the same connected component of a link graph, and so are all pointer references to the structure, no matter where they are in the program. The implication is that if two nodes of a link graph are not on the same connected component, i.e. there is no path between them, the corresponding pointer instances of SSA form are definitely not referencing the same linked data structure. Similarly, the same assertion can be obtained for pointer-induced aliases. That is, if between two nodes of a link graph there is no path which consists of ε edges only, the corresponding instances on SSA are not aliases.

Another special property is that the link graphs can identify where (at which program points) dynamic pointer-linked data structures are created and traversed. The information is carried by the directed edges: solid directed edges represent the creation and/or traversal of links between storage locations, whereas dashed ones means traversal down the links. If a linked data structure is created or traversed by a sequence of straight statements (i.e. not a loop), the link graph will be acyclic. Otherwise, those referenced nodes will be in the same strongly connected component, i.e. a cycle, with the iteration edge of the ϕ-function at the header of the loop (the other edge of the same ϕ is the initialization one). For example, the link graphs of the first loop that creates a linked structure and the second one that traverses it in Figure 2(a) are cyclic, as shown in Figure 2(b) and Figure 2(c), respectively.

One important ability is that link graphs can differentiate the cycles that represent cycles in run-time data structures from those that represent unbounded acyclic data structures, in contrast to other similar graphs, e.g. alias graphs [12], and storage shape graphs [2], etc. If a cycle of a link graph does not have an iteration edge of the ϕ-function at the header of the loop, then it signals a cycle in a data structure at runtime. Otherwise, if a cycle contains an iteration edge and does not include nodes which are simple fixed locations (e.g. scalar variables), it means an unbounded acyclic data structure. Of course, it is possible that a cyclic linked list can have unbounded number of nodes. In this case, the strongly connected component corresponding this cyclic structure will have two cycles. For example, if statements s0a and s9a are added to the program in Figure 2(a) to create a cyclic linked list, the link graph shown in Figure 3 has two cycles – one with the initialization edge of the ϕ-function at the header of the loop, while the other with iteration edge.

```
s0a   p1=> w1
s1    do while (...)
s2      w2= φ (w1, w3)
s3      allocate (x1)
s4      x1%car = 1
s5      x1%cdr => w2
s6      allocate (w3)
s7      w3%car = 2
s8      w3%cdr => x1
s9    end
s9a   p1%cdr => w2
```

(a) Program to Create Cyclic Linked List



(b) Link Graph

Figure 3: Cycles in Link Graph of Cyclic Linked List

4 Analysis Techniques Using Link Graphs

This section presents two algorithms of compile-time analysis, side effect analysis and DEF-USE information construction for programs with dynamic pointer-linked data structures.

4.1 Side Effect Analysis on Linked Data Structures

This algorithm detects side effects of programs with dynamic pointer-linked data structures. It reports the side effects to fixed locations, i.e. variables or allocated heap storage, of certain program points, blocks of code, or the whole program. It performs the analysis by collecting the side effects of statements from SSA form without concerning the aliases of pointers and linked data structures, and then propagates the information on link graphs. The algorithm works as follows:

1. Transform the program into SSA representation and build the corresponding link graphs.
2. Collect the set of instances (pointers or variables) from the SSA form that will be modified by assignment statements.
3. For each instance in the set, perform the following operations on the link graph:
   - Mark the side effect at the corresponding node on the link graph.
   - Starting from the node, mark all the nodes that can be reached through ε edges without passing any two ε links denoted by the same ϕ sign.

To give an example, this algorithm is applied on the program in Figure 2(a). The result will reveal that the side effects caused by statements s4 and s5 end up in the heap locations allocated by s3, and s7 and s8 by s6. Note that the side effects do not reach the instance w_1. It means the loop does not cause any side effects to any locations of the linked data structure starting from w_1, if it exists, that is created before entering the loop.

The side effect information can be more precise if different marks are used to represent various types of side effects. For example, modification to a field of a record can be denoted by placing a mark with the field name on the node of link graph. As a result, the precision of the side effect information is improved to record fields. Furthermore, the same link graph can be used to detect read-write conflicts by properly marking every read and write on the corresponding nodes. If there are any nodes on the link graph with conflicting marks, the function might have statements causing read-write conflicts.

4.2 DEF-USE Information of Dynamic Linked Data Structures

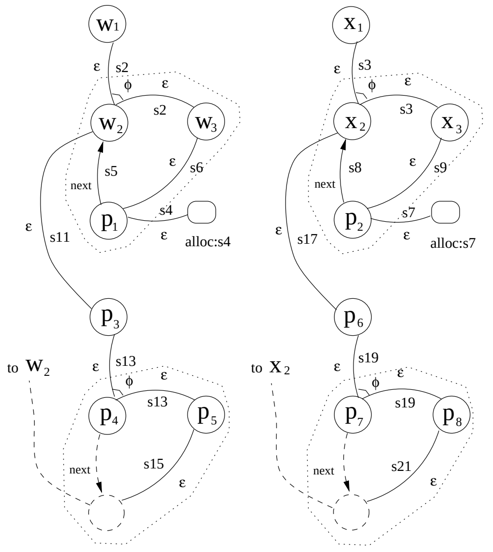
The DEF-USE chains of variables are defined for every statement, since the target location of each reference can be uniquely determined from program text. However, the same case might not apply on programs with pointers, because pointers can dynamically point to any locations. Furthermore, it is common that locations of nodes in dynamic linked data structures cannot be dereferenced through explicitly declared variable names, and consequently they have to be referenced through proper pointer initialization and link traversal. That is, references to linked data structures are usually performed by sequences of statements. Therefore, DEF-USE information of regions (blocks of statements), instead of statements, will be gathered.

```
s1      do while (...)
s2        w2= ϕ (w1, w3)
s3        x2= ϕ (x1, x3)
s4        allocate (p1)
s5        p1%next => w2
s6        w3=> p1
s7        allocate (p2)
s8        p2%next => x2
s9        x3=> p2
s10     end

s11     p3=> w2
s12     do while (...)
s13       p4= ϕ (p3, p5)
s14       ...
s15       p5=> p4%next
s16     end

s17     p6=> x2
s18     do while (...)
s19       p7= ϕ (p6, p8)
s20       ...
s21       p8=> p7%next
s22     end
```

(a) Program (in Extended SSA)



(b) Link Graph

Figure 4: DEF-USE Information of Linked Structures

This algorithm identifies the code fragments or program regions, especially loops, that create, modify, or traverse dynamic pointer-linked data structures, and gather the DEF-USE information among these code fragments. It works as follows:

1. Transform the program into SSA representation.
2. Build the corresponding link graphs without constructing virtual (dashed) undirected edges.
3. Traverse the link graphs and identify strongly connected components (cycles) with the iteration edges of the ϕ-functions at the headers of loops. Those cycles with solid directed edges represent loops that define, i.e. create or modify, the linked data structures, while those with virtual directed edges only are loops that traverse the linked structures without changing any connections.
4. Connect the virtual (dashed) undirected edges of link graphs. The DEF regions of a USE region are those that are connected to the USE region by undirected virtual links.
5. Identify the groups of statements that define and reference linked data structures by mapping the edges of the cycles on link graphs back to statements in SSA form.

Note that a program region that traverses a dynamic linked data structure might have more than one defining region since the data structure might be created by one region and modified by others before entering the current traversing region.

Applying the algorithm to the program in Figure 4(a) reveals that the first loop (s1-s10) in fact creates two independent linked lists starting from w and x respectively, while the second loop traverses the w list and the last loop references the other list. This information will be useful if the program is running on a (distributed) shared-memory multiprocessor system, since it recognizes that only a global synchronization is required right after the end of the first loop. There will be no reference conflicts between the second and the third loops because they traverse two independent linked data structures.

5 Interprocedural Analysis

This section describes the extensions of SSA representation and link graphs so that they can carry interprocedural information, and presents an example application of interprocedural compile-time analysis – safety analysis of user-defined reduction functions with dynamic pointer-linked data structures.

5.1 Interprocedural SSA Representation and Link Graphs

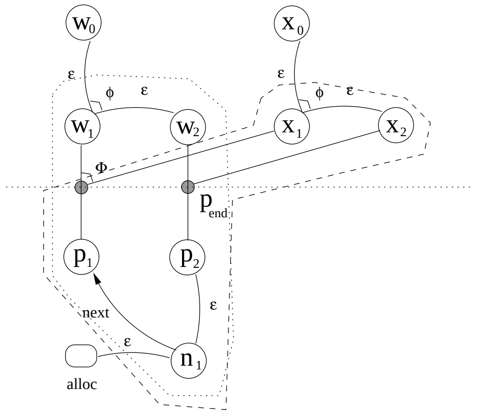
An extension to the SSA representation is required to model the procedure calls. It models the passing of parameters to and back from procedures. Each call-by-value parameter is modeled by assigning its actual parameter to formal parameter, whereas each call-by-reference parameter is modeled by two assignment statements – the first assigns the actual parameter to the formal one and the second assigns the formal parameter back to the actual one. Furthermore, since each procedure might be called at different call sites, a join function similar to ϕ-functions is needed to merge different incoming values of actual parameters and hence a Φ node is inserted for each parameter right before the procedure, if necessary. For example, a Φ-function is placed before the subroutine $insert$ in Figure 5(a), which has two call sites with actual parameters w and x, respectively. Each call site is attached with an assignment statement to get the value of formal parameter back to actual one. Note this extension applies to pointer SSA form as well.

```
do while (...)
  $w_1 = \phi(w_0, w_2)$
  call insert ($w_1$); $w_2$ => $p_{end}$
end do

do while (...)
  $x_1 = \phi(x_0, x_2)$
  call insert ($x_1$); $x_2$ => $p_{end}$
end do

$p_1 = \Phi(w_1, x_1)$
subroutine insert ($p_1$)
  allocate ($n_1$)
  $n_1$%next => $p_1$
  $p_2$=> $n_1$
end
$p_{end} = p_2$
```

(a) Interprocedural SSA



(b) Interprocedural Link Graph

Figure 5: Interprocedural SSA Representation and Link Graphs

Extra types of nodes need to be introduced to link graphs also in order to model procedure calls. They are placed right on the boundary that separates procedures. The boundary nodes with Φ-functions merge the incoming values from actual parameters, while the nodes without Φ model the return of values of formal parameters back to actual parameters. Consider the interprocedural SSA form shown in Figure 5(a). A boundary node with Φ is added to the link graph to copy the values of actual parameters w_1 and x_1 to the formal parameter p_1 and the boundary node p_{end} is included to return the final value of formal parameter back to actual parameters, as shown in Figure 5(b). This link graph depicts that two independent unbounded acyclic data structures are created by the two loops, which in turn call the same procedure $insert$ to insert new elements to the structures. This information can be obtained by traversing the link graph to find two cycles with ϕ-functions at headers of loops, if the link graph of the called procedure, i.e. $insert$, can be shared by the link graphs of calling procedures, but not the two edges of Φ boundary node.

5.2 Safety Analysis of User-Defined Reduction Functions

The introduction of user-defined reduction functions provides programmers the ability to formulate new combining operations on dynamic pointer-linked data structures in data-parallel languages [8, 17]. It usually takes two parameters – *reduction variable* is where the result is accumulated and *input parameter* passes the input value to the function. However, none have proposed algorithms to determine if user-defined reduction functions with dynamic pointer-linked data structures have undesired side effects. This section proposes an algorithm to determine if the side effects of reduction functions are safe.

The algorithm checks three conditions to determine if a user-defined reduction function is safe, i.e. it does not cause data access conflicts and the side effects will not change the final result. They are:

- The function does not cause any side effects to locations that are reachable from the input parameter,
- No side effects are imposed by the function to any variables referenced in the loop, and
- No statements in the loop, except for the one calling reduction function, reference the reduction variable.

The algorithm works as follows:

1. Construct the link graphs of the user-defined reduction function and the loop that encloses it.
2. Collect the side effects of the reduction function from the SSA representation, and propagate them on the link graphs by marking side effects on aliased nodes. Traverse the link graph of the second input parameter. If any nodes on the link graph have the side effect marks, the user-defined reduction function is considered as not safe.
3. Check all the nodes on the link graphs of the enclosing loop, except those of the reduction function and the calling statement. If there are any nodes marked by the side effects, the user-defined reduction function is considered as not safe. Otherwise, each traversed node is marked as a reference, and the mark is propagated to aliased nodes.
4. Traverse the link graph of the reduction variable. If any nodes on the link graph have reference marks, enclosing loop is considered as not safe. Otherwise, the reduction operation is considered as safe.

Consider the example user-defined reduction in Figure 6. It is an independent loop to create an unordered linked list by using two different versions of user-defined reduction function $insert()$. It is easy to tell that version 1 is not safe since it modifies the input parameter $node$ directly, whereas version 2 does not causes side effects on $node$ and it can be proved safe by applying the algorithm above. However, there are cases that user-defined reduction functions are not safe even though they do not seem to cause any side effects to input parameters. For example, both versions of the append function in Figure 7 do not modify the input parameter $node$ directly. The safety analysis algorithm reveals that version 1 is not safe since it modifies the node appended to the list at the previous iteration, while version 2 is safe.

```
{Main Loop}                  {Version 1}                   {Version 2}
!HPF$ INDEPENDENT            subroutine insert (list, node)   subroutine insert (list, node)
do i = 1, N                    node%next ⇒ list               allocate (p)
  · · ·                        list ⇒ node                    p%node ⇒ node
  !HPF$ REDUCE               end                              p%next ⇒ list
  call insert (list, node(i))                                 list ⇒ p
end do                                                      end
```

Figure 6: User-Defined Reduction – Node Insertion

```
{Main Loop}                  {Version 1}                       {Version 2}
!HPF$ INDEPENDENT            subroutine append (list, node)    subroutine append (list, node)
do i = 1, N                    p ⇒ list                          p ⇒ list
  · · ·                        do while (associated(p%next))     do while (associated(p%next))
  !HPF$ REDUCE                   p ⇒ p%next                        p ⇒ p%next
  call append (list, node(i))  end do                            end do
end do                         p%next ⇒ node                     allocate (p%next)
                             end                                 p%next%node ⇒ node
                                                               end
```

Figure 7: User-Defined Reduction – Node Append

6 Example

This section studies the properties of the linked list and tree used in a simplified version of the Barnes-Hut tree code [1] by building link graphs and performing the side effect analysis and DEF-USE information construction. Based on the properties reported by the analysis techniques, the Barnes-Hut tree code has been parallelized by hand and the timing results of the hand-parallelized tree code on Stanford DASH and Intel Paragon will be presented.

Figure 8 shows the simplified version of the tree code. It consists of two loops – the first loop traverses the particle list and creates a binary tree, and the second one calculates the force imposed on each particle by all other particles by calling a recursive tree traversal function. The link graphs of the simplified Barnes-Hut tree code are shown in Figure 9.

By observing the link graphs and applying the algorithms in the previous section, the following information can be obtained by compilers:

- The tree creation loop (S1–S24) creates an acyclic linked data structure. The strongly connected component of the link graph in Figure 9(a) contains the iteration edge of loop header, and hence concludes the created linked data structure is acyclic.

- The particle list and the tree are connected. However, paths exist only from the tree to the particle list, but not vice versa. This connection causes other shape analysis techniques to wrongly guess the tree is cyclic [9].

- Applying the DEF-USE algorithm will reveal that linked data structure referenced in the second loop (S25–29) is created by the first loop. The other linked structure is traversed by both loops and and its structure remains intact.

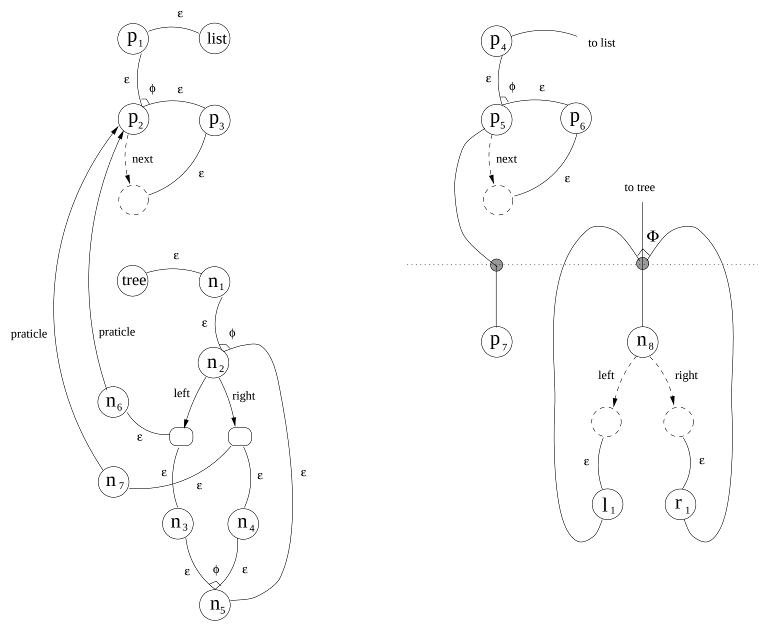
- The side effect analysis algorithm will detect the side effects that occur to the allocated nodes during tree creation loop and to the nodes of the particle list in the tree traversal loop. The tree traversal loop does not incur any

```
      {Tree Creation}
S1    p ⇒ list
S2    do while (associated(p))
S3      n ⇒ tree
S4      do
S5        if (COMPARE(p, n)) then
S6          if (associated(n%left)) then
S7            n ⇒ n%left
S8          else
S9            allocate(n%left)
S10           n%left%particle ⇒ p
S11           exit
S12         end if
S13       else
            same as S6-S12, replace left by right
S21       end if
S22     end do
S23     p ⇒ p%next
S24   end do
```

```
      {Tree Traversal}
S25   p ⇒ list
S26   do while (associated(p))
S27     p%force = traverse_tree(p, tree)
S28     p ⇒ p%next
S29   end do

S30   recursive function traverse_tree(p, tree)
S31     if (TOO_FAR_AWAY(p, tree) then
S32       traverse_tree = FORCE(p, tree)
S33     else
S34       traverse_tree = traverse_tree(p, tree%left)
                    + traverse_tree(p, tree%right)
S35     end if
S36     return
S37   end
```

Figure 8: Simplified Barnes-Hut Tree Code



(a) Tree Creation (b) Tree Traversal

Figure 9: Link Graphs of Tree Code

side effects to tree. The implication of this information is that the tree can be traversed independently by any particles in the list, since each iteration only updates the force field of the particle currently being referenced.

Based on the information shown above, the tree traversal loop can be executed in parallel. That is, each particle can traverse the tree independently. However, the tree creation loop is executed sequentially. The timing results (in seconds) of hand-parallelized Barnes-Hut code on Stanford DASH are listed in Table 1. The tree code was executed for the time span of 0.5 seconds with each time step 0.025 seconds, and consequently 21 times of tree creations and traversals were performed. It shows speedup of 5.4 on 8 processors and 7.6 on 16.

Table 1: Execution Tims for Barnes-Hut on DASH

Number of Particles	Processors				
	1	2	4	8	16
8192	4024.98	2093.03	1104.62	731.65	517.14

The tree code can be parallelized on distributed memory machines in the same way. However, since the particles are distributed over processors, each processor has to collect particles from all other processors to construct the tree. Once the tree construction is finished, each processor iterates through local particles and traverses the tree independently. Although interprocessor communications incur overheads, good locality is observed and hence better speedup is achieved. It achieves speedup of 13 on 16 processors of Intel Paragon. Table 2 shows the execution times of the hand-parallelized tree code with 8K and 16K particles.

Table 2: Execution Tims for Barnes-Hut on Paragon

Number of Particles	Processors				
	1	2	4	8	16
8192	1007.65	510.20	263.04	138.76	76.73
16384	2288.66	1157.86	592.40	310.65	169.83

The tree creation phase of both parallelized versions can not be executed in parallel because the original program is designed to create a single tree structure to store particle information. As a result, compilers would have to sequentialize the tree creation phase. However, other researchers have proposed and implemented tree codes that processors create subtrees from local particles in parallel [14]. Therefore, both phases can be executed in parallel. The user-defined reduction functions can be used to program this type of codes in data-parallel languages.

7 Related Work

The analysis done in this paper is closely related to the side effect analysis on programs with pointers [3, 7, 11] and conflict/interference analysis for programs with dynamic pointer-linked data structures [10, 12]. Each of these algorithms builds either tables, graphs, or sets to represent the aliases or connections of locations, and then performs compile-time analysis on these auxiliary structures. The difference of this work from others is that a single link graph is constructed for each procedure or a block of code, whereas others have to build an auxiliary structure for every statement.

Another related field is shape analysis, which is to find the characterization of possible shapes of the dynamic linked data structures in programs [2, 9, 15, 16]. The link graphs in this paper can differentiate the cycles that represent cycles in run-time data structures from those that represent unbounded acyclic data structures, while the Storage Shape Graphs (SSG) can not [2]. Plevyak et al. adapted the SSA and developed the Abstract Storage Graph (ASG) that solved about problem [15]. However, unlike link graphs which can be constructed from SSA in a straightforward manner, it takes considerable efforts to construct an ASG and, furthermore, it might not be easy to compare ASGs of different statements to determine how the dynamic linked structures are modified because of possible deconstruction and compression operations on them. The shape analysis algorithm proposed by Ghiya and Hendren [9] reports that linked structures created by the n-body solver [1] are cyclic, whereas the link graphs reveal that it is acyclic. The link graphs are, in a sense, similar to the shape-graphs developed by Sagiv et al [16]. The nodes of the shape-graphs represent distinct locations while those of link graphs correspond to unique pointer instances, and the edges of both type of graphs depict the connections. However, since each shape-graph records the current condition of linked structures at a program point, it is not easy to characterize the effects by the execution of statements on the linked structures. On the other hand, link graphs can not handle destructive updating on links as well as shape-graphs, since they are built based on the information of SSA representation.

The distinct feature of this work is that link graphs can depict the process of creation and traversal of dynamic pointer-linked data structures by sequences of statements, and consequently DEF-USE information of these linked structures can be exploited for optimizations or program transformations by compilers. This piece of information is not easy to extract from other auxiliary structures proposed by other researchers.

Recently, commutativity analysis has been proposed as a new framework for parallelizing compilers [13]. It parallelizes programs by analyzing if the operations commute. However, it only applies to object-based programs. Furthermore, its parallelization approach tends to generate programs with fine-grained parallelism and hence the generated code might not be efficient when running on distributed memory systems.

8 Summary

This paper proposed link graphs to portray the alias information between pointers and the connection relationships of dynamic pointer-linked data structures. Algorithms were presented to demonstrate that information essential for optimizations on sequential and data-parallel programs with dynamic pointer-linked data structures can be gathered by using the link graphs as the basis.

The extended form of SSA representation and link graphs have been implemented on top of the ParaScope programming environment, originally developed by Rice University [4]. The implementation of the algorithms proposed in the paper are currently underway. The algorithms will be integrated as part of Fortran 90D compiler that transforms data-parallel programs with dynamic pointer-linked data structures into efficient parallel code.

Although the examples in this paper are programmed in Fortran 90, which only supports single level pointers, multiple level pointers in other languages, e.g. C, can also be handled and be represented by SSA form. The key concept is to treat each level as a record that contains a single selector (or field), which carries the location of the record of next level pointer.

References

[1] J. Barnes and P. Hut. A hierarchical O(NlogN) force-calculation algorithm. *Nature*, pages 446–449, December 1976.

[2] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *SIGPLAN Notices*, 25(6):296–310, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

[3] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, January 1993.

[4] K. D. Cooper, M. W. Hall, R. Hood, K. Kennedy, K. McKinley, J. Mellor-Crummey, L. Torczon, and S. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, pages 244–263, February 1993.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[6] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. *SIGPLAN Notices*, 28(6):36–45, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[7] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural Points-to analysis in the presence of function pointers. *SIGPLAN Notices*, 29(6):242–256, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[8] High Performance Fortran Forum. *HPF-2 Scope of Activities and Motivating Applications*, November 1994.

[9] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of POPL '96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, St. Petersburg Beach, Florida, January 1996.

[10] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[11] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural side effect analysis with pointer aliasing. *SIGPLAN Notices*, 28(6):56–67, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[12] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. *SIGPLAN Notices*, 23(7):21–34, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[13] Pedro C. Diniz Martin C. Rinard. Commutativity analysis: A new analysis framework for parallelizing compilers. *SIGPLAN Notices*, 31(5):54–67, May 1996. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.

[14] Kevin M. Olson and Charles V. Packer. An n-body tree algorithm for the Cray T3D. Technical Report 199882, NASA Contractor Report, May 1996.

[15] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 37–56, Portland, Oregon, August 1993. Lecture Notes in Computer Science, Vol. 768, Springer Verlag.

[16] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of POPL '96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, St. Petersburg Beach, Florida, January 1996.

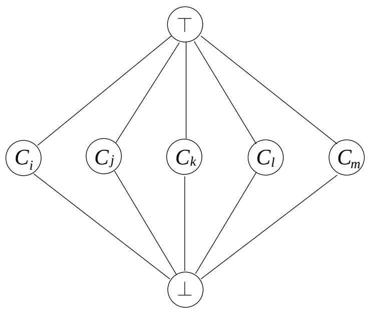
[17] Guhan Viswanathan and James R. Larus. User-defined reductions for communication in data-parallel languages. Technical Report 1293, Computer Science Department, University of Wisconsin-Madison, January 1996.

[18] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[19] Michael Wolfe. Beyond induction variables. *SIGPLAN Notices*, 27(7):162–174, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.

A Constant Propagation on Programs with Pointers

This section presents a constant propagation algorithm to demonstrate that it is easy to adapt existing algorithms to perform analysis on programs with pointers using the extended SSA form as the basis. This algorithm is adapted from the Sparse Conditional Constant (SCC) algorithm by Wegman and Zadeck [18]. It examines the SSA form at most twice, in contrast to the one proposed by Cytron and Gershbein [6] which constructs a sequence of approximating and partial SSA forms until convergence is reached.



(a) Three Level Lattice

$$\text{any} \sqcap \top = \text{any}$$
$$\text{any} \sqcap \bot = \bot$$
$$C_i \sqcap C_j = C_i \quad \text{if } i = j$$
$$C_i \sqcap C_j = \bot \quad \text{if } i \neq j$$

(b) $\sqcap$ Operation

Figure 10: The Lattice and Rules for $\sqcap$ Operator

This algorithm proceeds by lowering the *Lattice Cell* of each node until a fixed point is achieved. The lattice and the rules of the meet ($\sqcap$) operator are depicted in Figure 10. It uses three worklists: *Flow WorkList* is a worklist of program flow graph, *SSA WorkList* is a worklist of variable SSA edges, and *Pointer WorkList* is a worklist of pointer SSA edges. This algorithm works as follows:

1. Initialize the Flow WorkList to contain the edges exiting the start node of the program. The SSA WorkList and Pointer WorkList are initially empty.
   Each program flow graph edge has an associated flag, the *Executable Flag*, that controls the evaluation of ϕ-functions in the destination node of that edge. This flag is initially `false` for all edges.
   Each node of the variable SSA has a Lattice Cell, which is initially $\top$, while every pointer SSA node has a Lattice Set, which is initially empty.

2. Halt execution when all three worklists become empty.
   Pointer WorkList has the priority.
   Execution may proceed by processing items from either Flow WorkList or SSA WorkList when Pointer WorkList is empty.

3. If the item is a program flow graph edge from the Flow WorkList, then examine the Executable Flag of that edge. If the Executable Flag is `true` do nothing; otherwise:

   (a) Mark the Executable Flag of the edge as `true`.

   (b) Perform Visit-ϕ for all of the ϕ-functions at the destination node.

   (c) If only one of the Executable Flags associated with the incoming program flow graph is `true` (i.e., if this is the first time this node has been evaluated), then perform Visit Expression for the expression in this node and perform Update Alias if the corresponding node exists.

   (d) If the node only contains one outgoing flow graph edge, add that edge to the Flow WorkList.

4. If the item is an SSA edge from either the SSA WorkList or Pointer WorkList and the destination of that edge is a ϕ-function, perform Visit-ϕ.

5. If the item is an SSA edge from the SSA WorkList and the destination of the edge is an expression, the examine Executable Flags for the program flow edges reaching that node. If any of them are `true`, perform Visit Expression. Otherwise, do nothing.

6. If the item is an SSA edge from the Pointer WorkList and the destination of the edge is an expression, the examine Executable Flags for the program flow edges reaching that node. If any of them are `true`, perform Update Alias. Otherwise do nothing.

Visit-ϕ is defined as follows: The Lattice Cell, if the SSA node does not have any extra reaching definitions, or Lattice Set for each operand of the ϕ-function are defined on the basis of the Executable Flag for the corresponding program flow edge.

- `executable`: The Lattice Cell has the same value as the Lattice Cell at the definition end of the SSA edge. The Lattice Set, if such pointer SSA node exists, has the same set of elements as the Lattice Cell at the definition end of the pointer SSA edge.
- `not-executable`: The Lattice Cell has the value $\top$. The Lattice Set is $\emptyset$, if exists.

If any operands have extra reaching definition edges, the values of the Lattice Cells will be the meet of the values of reaching definitions with executable program flow edges. The value of of Lattice Cell (Lattice Set) associated with the output of a ϕ-function of variable SSA (pointer SSA) is defined to be the meet (union) of all arguments whose corresponding in-edge has been marked executable.

Visit Expression is defined as follows: Evaluate the expression obtaining the values of the operands from the Lattice Cells where they are defined (meet operation is performed if extra reaching definitions exist) and using the expression rules defined in Figure 10. If this changes the value of the Lattice Cell of the output of the expression, do the following:

1. If the expression is part of an assignment node, add to the SSA WorkList all SSA edges starting at the definition for that node.

2. If the expression controls a conditional branch, some outgoing flow graph edges must be added to the Flow WorkList. If the Lattice Cell has value $\perp$, all exit edges must be added to the Flow WorkList. If the value is a constant, only the flow graph edge executed as the result of the branch is added to the Flow WorkList.

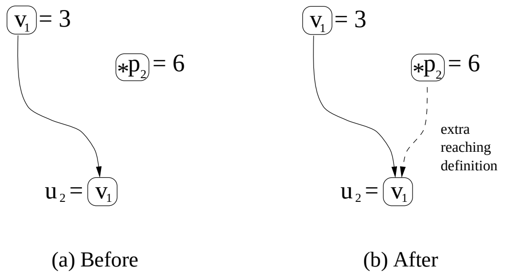


Figure 11: An Extra Reaching Added Due to Alias

Update Alias is defined as follows: Calculate the alias sets of the operands from the Lattice Sets where they are defined. If this changes the value of the Lattice Set of the output of the expression, do the following for each newly introduced alias:

- Follow the dashed link of the expression from pointer SSA to variable SSA and start from the second definition, and add an extra reaching definition link to every use of the current definition of the aliased variable. For example, the assignment statement $\star p_2 = 6$ causes an extra reaching definition to the use of v in the statement $u_2 = v_1$, as shown in Figure 11, since that p is aliased to v by the pointer assignment statement $p_1 \Rightarrow v$.