Side Effect Analysis on User-Defined Reduction Functions 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

Abstract

This paper presents an algorithm to analyze the side effects of user-defined reduction functions with dynamic pointer-linked data structures. It uses the static single statement (SSA) intermediate form and an auxiliary structure, *link graph*, to represent alias information of the whole procedure. Due to the static associations of pointers in SSA form, the side effects can first be collected without concerning aliases and then be propagated on the link graph to gather final side effect information.

1 Introduction

Reduction operations are an important source of parallelism in scientific computational applications, since they can be efficiently executed in parallel due to the special property that the result of a reduction on a set of data objects will be the same regardless the order which it is evaluated. Researchers have been able to achieve high performance by exploiting this parallelism. In order to facilitate the use of reduction operations, many data-parallel languages and compilers support predefined associative arithmetic and logical reduction operators, such as summation and logical-AND, etc. [9, 15] Meanwhile, more general forms of reduction operations with user-defined operators have also been proposed and implemented in compilers [8, 16]. The introduction of user-defined reduction functions provides programmers the ability to formulate new combining operations on scalars, arrays, compound data types, and even dynamic pointer-linked data structures.

The predefined reduction operators in data-parallel languages are side-effect free. However, it is not guaranteed all user-defined reduction functions will share this property. Some languages require that user-defined reduction functions must be declared as *pure* functions [8], while others rely on the guarantee from users that the functions are safe [16]. The former is too restrictive on programming and might cause inefficiency for reduction operations, whereas the latter provides no help to users from compilers even for the cases that side effect analysis reveals that user-defined reduction might not be safe.

This paper presents a simple side analysis analysis algorithm to determine the side effects to input parameters of user-defined reduction functions operating on dynamic pointer-linked data structures. It is closely related to the compile-time conflict/interference analysis for programs with pointers [11, 13], and side effect analysis [2, 3, 12]. This algorithm has three steps. Each generalized reduction function is first transformed to SSA (Static Single Assignment) intermediate representation [4]. A *link graph* will then be constructed to portray the connections among the pointer instances of SSA form. Finally, the side effects collected from the SSA form will be propagated on the link graph to determine the side effects caused by the pointer aliases.

This paper is organized as follows. Section 2 defines the reduction operations and discusses the semantics restriction on user-defined reduction operators. The side effect analysis algorithm is presented in Section 3. Section 4 surveys and compares related work. Section 5 summarized this paper and sketches future research directions.

2 Reductions

This section defines the user-defined reduction operations, and discusses the semantics restrictions imposed on them.

2.1 Definition

A reduction operation can be informally defined as an accumulation of elements of a vector, where the (binary) accumulating operator is associative and commutative. For example, the summation of the N elements of a vector x can be performed by the following Fortran loop

```
do i = 1, N
   sum = sum + x(i)
end do
```

In a loop that performs a reduction, the output dependencies between different iterations can be ignored, thus enabling parallelization.

Formally, a reduction operation is the execution of a binary reduction operator f on a collection of arguments C. It returns the result of combining the data elements $d_1\ d_2\ d_3\ \cdots\ d_n$ of C with f:

$$reduce_f(d_1\ d_2\ \cdots\ d_n) \equiv f(d_1,\ f(d_2,\ \cdots,\ f(d_{n-1},\ d_n)\cdots))$$

The operator f is required to be associative and commutative to guarantee that the result of a reduction operation will be the same regardless of the order of execution of the operator on the elements of C. This property makes reduction operations an important source of parallelism for scientific computational applications to achieve good performance.

2.2 User-Defined Reduction

Most data-parallel languages and compilers only support predefined associative arithmetic and logical operators, such as summation and logical-AND, etc. [9, 15] In order to provide programmers more flexibility in defining various reduction operations, user-defined reduction functions have been proposed [16]. There also have been discussions on this issue by the on-going HPF-2 project [8].

The reduction operators can be any user-defined functions as long as they obey the same semantics and restrictions of the built-in ones. This feature provides programmers the ability to formulate new combining operations on scalars, compound data types, and even dynamic pointer-linked data structures. For example, a user-defined reduction function can be defined to append a node to a linked list. Such reduction operator can then be used in a loop to build a linked list, as shown in Figure 1. Note that the example is presented in Fortran 90 with the directives considered by HPF-2 project.

2.3 Semantics and Language Support

The predefined reduction operators in data-parallel languages are side-effect free. In other words, each operator takes the values of operands and returns the result of the computation, without modifying the contents of operands and all other variables. However, it is not guaranteed all user-defined reduction functions will share this property. Therefore,

```
!HPF$   INDEPENDENT
        do i = 1, no-nodes
          . . .
!HPF$     REDUCE
          call append (list, node(i))
        end do
```

Figure 1: Loop to Append Nodes to a List

some languages require that user-defined reduction functions must be declared as *pure* functions [8]. A function is pure if it does not change the values, pointer associations, or data mapping of any of its arguments or global variables [7]. The only effect of a pure function on program execution is to return a result. As a result, if the input parameters and returned result of a pure reduction function are complex pointer-linked data structures, the whole structures of the input parameters might have to be copied to produce the result. That means every invocation of this pure reduction function with linked data structures incurs copying of the whole input structures and hence it would cause inefficiency.

In order to avoid inefficiency, other languages allow side effects but rely on the guarantee from programmers that the functions are *safe* [16]. A reduction function is safe when its side effects will not affect the result of computation, regardless of the order the iterations of the loop surrounding it are executed. This approach avoids the task of side effect analysis on functions with dynamic pointer-linked data structures, but it provides no help to users from compilers even for the cases that side effect analysis reveals that user-defined reduction might not be safe. Consequently, this strategy is prone to mistakes since programmers might provide unsafe reduction functions due to the lack of the knowledge of possible side effects. Therefore, side effect analysis should be performed on user-defined reduction functions even with pointers to reduce the possibility of mistakes.

To demonstrate that the side effect information will help users identify unexpected side effects, two versions of the list append functions shown in Figure 2 will be used as the examples for side effect analysis. Two different lists generated by the two versions of list append function are shown in Figure 3. At first glance, neither version incurs side effects to the input parameter $node$, since $node$ is not directly modified in the functions. However, this observation is not valid for the version (a), because that once $node$ is appended to the list it might be modified during the later invocation of the same function. On the contrary, version (b) will not incur side effects on the argument $node$ since the modification is limited to internal nodes allocated in the function. Therefore, users should be notified that side effects might occur on the set of nodes to be linked to a list. The algorithm presented in the next section will perform side effect analysis on function with dynamic pointer-linked data structures, and capture this subtle type of side effects.

3 Side Effect Analysis

This section presents an algorithm to perform compile-time analysis to summarize the side effects to input parameters of user-defined functions.

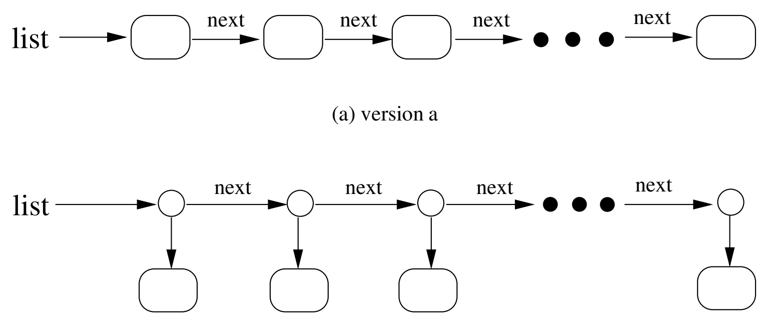
3.1 Algorithm

The algorithm to analyze side effects of a reduction function can be broken into three steps:

1. Transform the function into an SSA representation. Every pointer variable in the original program is replaced by several new variables such that there is only one assignment to each new variable.

(a)	(b)
ptr ⇒ list	ptr ⇒ list
do	do
if (.not. associated(ptr%next)) exit	if (.not. associated(ptr%next)) exit
ptr ⇒ ptr%next	ptr ⇒ ptr%next
end do	end do
ptr%next ⇒ node	allocate(ptr%next)
	ptr%next%node ⇒ node

Figure 2: Two Versions of List Append Function `append(list,node)`



(a) version a

(b) version b

Figure 3: Results of Two List Append Functions

2. Create an auxiliary structure called the *link graph* that represents the connection among pointer variables. Each node in the link graph corresponds to a pointer variable in the SSA form.

3. Determine side effects on the nodes of the link graph. For each assignment that causes side effects on a node in the link graph, mark the side effects on the nodes that are reachable following the links.

3.1.1 Creating SSA Form on Pointers

A program is defined to be in SSA (Static Single Assignment) form if, for every original variable V, trivial merging functions, ϕ-functions, for V have been inserted and each mention for V has been changed to mention of a new name V_i such that the following conditions hold:

- If a program flow graph node Z is the first node common to two non-null paths $X \stackrel{+}{\to} Z$ and $Y \stackrel{+}{\to} Z$ that start at nodes X and Y containing assignments to V, then a ϕ-function for V has been inserted at Z.
- Each new name V_i for V is the target of exactly one assignment statement in the program.
- Along any program flow path, consider any use of a new name V_i for V (in the transformed program) and the corresponding use of V (in the original program). Then V and V_i have the same value.

Although the above format is originally designed as an intermediate representation for programs without pointers, the same transformation can be also performed on pointer variables. For instance, Figure 4 presents the list append functions shown in Figure 2 in SSA form.

(a)		(b)	
S1	$\text{ptr}_1 \Rightarrow \text{list}_1$	S1	$\text{ptr}_1 \Rightarrow \text{list}_1$
S2	do	S2	do
S3	$\text{ptr}_2 = \phi\ (\text{ptr}_1, \text{ptr}_3)$	S3	$\text{ptr}_2 = \phi\ (\text{ptr}_1, \text{ptr}_3)$
S4	if (.not. associated(ptr_2%next)) exit	S4	if (.not. associated(ptr_2%next)) exit
S5	$\text{ptr}_3 \Rightarrow \text{ptr}_2\%\text{next}$	S5	$\text{ptr}_3 \Rightarrow \text{ptr}_2\%\text{next}$
S6	end do	S6	end do
S7	$\text{ptr}_2\%\text{next} \Rightarrow \text{node}_1$	S7	allocate(ptr_2%next)
		S8	$\text{temp}_1 \Rightarrow \text{ptr}_2\%\text{next}$
		S9	$\text{temp}_1\%\text{node} \Rightarrow \text{node}_1$

Figure 4: List Append Functions in SSA Form

The special property of the SSA form is that every use of any variable has only one reaching definition. It implies that the definition to every variable is static, i.e. it will not be affected by any other assignment statements in the same program. Original programs are transformed to SSA form by renaming – giving a unique name to each assignment to a variable. When the same transformation is performed on pointers (pointer variables), each pointer is linked to memory locations by a pointer assignment statement only. Consequently, the links between pointers will be static through the whole program. This property will simply the analysis process.

A separate SSA representation will be built solely on pointers. Pointer variables are transformed in the same way as variables in regular SSA form since each pointer variable links to a memory location can be viewed as a regular variable carries a value. 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

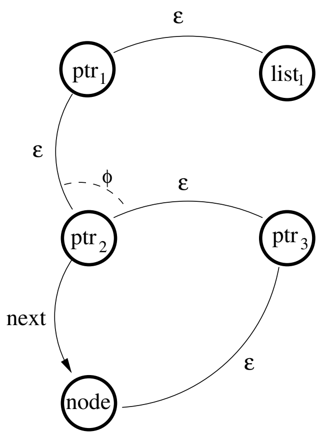
1. $ptr_1 \Rightarrow node_1$
2. $ptr_1 \Rightarrow node_1\%next$
3. $ptr_1\%next \Rightarrow node_1$
4. $allocate\ (ptr_1)$
5. $ptr_3 = \phi\ (ptr_1,\ ptr_2)$

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.

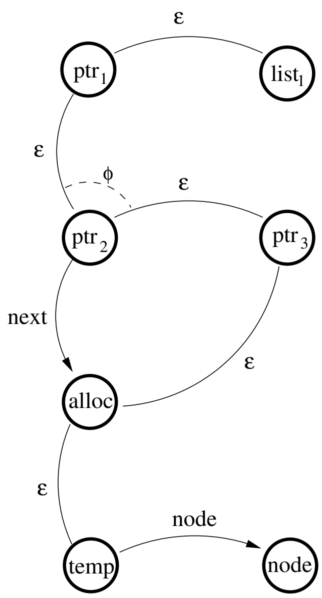
Although the examples in this paper is programmed in Fortran 90, which only supports single level pointers, multiple level pointers in other languages, e.g. C, can also be handled and be represented by SSA form. The key concept is to portray each level as a record that contains a single selector (or field), which carries the location of the record of next level pointer. For example, a two-level pointer $_{**}p$ (in C) is represented as a pointer to a record that has a single selector with the value $*p$. Therefore, a reference by a multiple level pointer can be transformed to a reference to a location through a sequence of selectors starting from the pointer.

3.1.2 Building Link Graphs

Once the pointer SSA form of a reduction function is constructed, the connection between of pointers in the function can be represented by a *link graph*. For instance, Figure 5 demonstrates the link graphs built from the SSA in Figure 4. Each node in the graphs corresponds to an instance of a pointer on SSA representation. Every link in the graph between two nodes means that the two pointer instances might point to the same memory location caused by a pointer assignment statement that defines either instance.



(a) Version a



(b) Version b

Figure 5: Constructing A Tree on Two Processors

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

- Construct a node for each pointer instance on SSA. Similarly, a node is created for each allocation statement.
- For each type 1 statement, i.e. $ptr_1 \Rightarrow node_1$, create a undirected ε edge between node ptr_1 and $node_1$. However, two ε edges will be generated by each statement with a ϕ function. For example, the statement $ptr_3 = \phi\ (ptr_1,\ ptr_2)$ causes ε links built 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.
- For each type 3 statement, $ptr_1\%next \Rightarrow node_1$, build a directed link with label $next$ from ptr_1 to $node_1$.
- For every type 2 statement, $ptr_1 \Rightarrow node_1\%next$, 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.

3.1.3 Determining Side Effects

The set of pointer variables that will be modified by assignment or pointer assignment statements is collected from the SSA form. For each pointer 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 node that can be reached through ε edges without passing any two ε links denoted by a ϕ sign.

The side effects of the functions are the memory locations pointed by the pointers with marked nodes.

Note that the side effect information can be more precise if different marks are used to represent various type 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.

Consider function (a) in Figure 4. The only statement that causes side effects is S7, $ptr_2\%next \Rightarrow node_1$, which modifies the $next$ field of the record pointed by ptr_2. Consequently, side effects might occur on the locations pointed by pointer instances that are reachable from ptr_2 node through ε links without passing any two ε links denoted by a ϕ mark. Therefore, following the ε links on the link graph of Figure 5(a), both parameters $list$ and $node$ of the append function might be modified by S7.

The same analysis process can be applied on the program in Figure 4(b) and its corresponding link graph in Figure 5(b). The pointer assignment statement S9 might modify the parameter $list$ and the memory locations allocated by statement S7. The second parameter of list append function, $node$, is free from side effects in this function.

3.2 Discussion

This algorithm is designed specifically for side effect analysis for user-defined reduction functions operating on dynamic pointer-linked data structures. It takes advantage of the fact that in every reduction loop the same reduction function will be repetitively executed with the same reduction variable, where the dynamic pointer-linked data structure is created and stored. Consequently, the side effects of a reduction function can be analyzed by using the link graph.

This algorithm can be used for safe analysis for user-defined reduction functions if a function is considered safe when it only modifies the reduction variable and does not change values of any input arguments or global variables. Similarly, a reduction function with dynamic pointer-linked data structures is safe if it does not change the contents of any memory locations that are reachable through input arguments or global variables, except the reduction variable. This should be a reasonable semantics restriction based on the following observations:

- The iterations of the loop that encloses a reduction function can be executed in any order. If a reduction function modifies the values of any input parameters or global variables, it might sequentialize the loop execution.
- The reduction variable is the place to accumulate the result of reduction operation. Before the execution of the loop enclosing the function is over, the value of the reduction variable is undefined since the iterations can be executed in any order. The reduction function call will the only statement that references the reduction variable in the loop.

This safe analysis is not precise due to the limitation of this algorithm. Currently the link graphs do not carry interprocedural information. Therefore, it only detects the possible side effects on the reduction variable and the input parameter of each reduction function. It does not detect if any variables referenced in reduction loop are modified by reduction functions. The investigation of interprocedural analysis is under way.

In this paper, link graphs are constructed mainly to collect all possible side effects due to the presence of pointers. However, they carry information which can be exploited for other purposes. One immediate application is to represent how a dynamic pointer-linked data structure is created and modified. It can be done because the the statements in the SSA corresponding to directed edges of the link graph are exactly where the links of data structure are modified. The ε links only account for references to the structure. Therefore, the DEF-USE information of dynamic data structures can be extracted from link graphs.

4 Related Work

Predefined reduction operators are commonly supported by sequential and data-parallel languages [9, 15]. User-defined reduction operations are being considered by the on-going HPF-2 project [8] and by researchers [16], to provide programmers more flexibility in defining various reduction operations. Since the user-defined reduction operators in HPF-2 proposal are required to be pure and those in other languages [16] are assumed to be safe, they are not ideal for users to program reduction operations on dynamic pointer-linked data structures. This paper presents an algorithm to detect the information of possible side effect of user-defined reduction functions to help programmers identify possible mistakes or to provide compilers the information to perform safe analysis if semantics of safe functions is defined.

This paper relies on explicit declaration of reduction functions by programmers. Fortran 90 with proposed HPF directives is used the target language. Automatic reduction recognition is beyond its scope. However, there have been efforts on this topic, though only automatic recognition of reduction operations on scalars and arrays has been studied [10, 14]. Once reduction operations are identified, either through explicit declarations or by automatic recognition, compilers can transform sequential or data-parallel programs into efficient SPMD (Single Program Multiple Data) code [5, 6].

Previous proposed algorithms for side effect analysis are targeting on Fortran-like languages [1, 3]. The analysis done in this paper is closely related to the side effect analysis [2, 12] and conflict/interference analysis [11, 13] for programs with dynamic pointer-linked data structures. Each of these algorithms creates graphs or tables to represent the connections (or aliases) between pointers. The conflict/interference analysis algorithms target on possible read-write conflicts or interferences between any two statements for the purpose of parallelization. Consequently, an alias graph or path table is constructed for every statement and the side effect information is gathered from the alias graphs or path tables. Similarly, the side effect analysis algorithms are performed after the alias information for each statement is collected. The side effects of each statement is analyzed based on the alias information and are accumulated to produce the side effects of procedures or blocks of code. However, the most significant difference of this algorithm from others is that a single graph is constructed for each procedure in this paper whereas others have to build a table or graph for every statement. The side effects of a procedure or a block of code are first gathered without concerning the aliases, and then they are propagated through the links of the link graph.

5 Summary and Future Work

This paper presents an algorithm to perform side effect analysis on the reduction variable and the input parameter. This information can be used for conservative safe analysis on user-defined reduction functions with dynamic pointer-linked data structures. If the side effects are contained in the reduction variable, the reduction function is safe. Otherwise, if it considered not safe if the reduction function might modify the value of the input parameter.

Future work of this research will be as follows:

- Extend the algorithm to cover interprocedural side effect analysis.
- Development an efficient algorithm to gather interprocedural DEF-USE information of dynamic pointer-linked data structures.
- Implement a prototype data-parallel compiler to transform programs with generalized reduction functions to efficient SPMD code.

References

[1] John Banning. An efficient way to find side effects of procedure calls and aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, San Antonio, Texas, January 1979.

[2] 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.

[3] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. *SIGPLAN Notices*, 23(7):57–66, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[4] 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.

[5] Raja Das, Joel Saltz, and Reinhard von Hanxleden. Slicing analysis and indirect access to distributed arrays. In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 152–168. Springer-Verlag, August 1993. Also available as University of Maryland Technical Report CS-TR-3076 and UMIACS-TR-93-42.

[6] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[7] High Performance Fortran Forum. High performance fortran language specification, version 1.1. Technical Report CRPC-TR92210, Center for Research on Parallel Computation, November 1994.

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

[9] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

[10] Mary W. Hall, Saman P. Amarasinghe, Brain R. Murphy, Shih-Wei Lian, and Monica S. Lam. Detecting coarse-grain parallelism using an interprocedural parallizing compiler. In *Proceedings of Supercomputing'95*, San Diego, CA, December 1995.

[11] 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.

[12] 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*.

[13] 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*.

[14] Bill Pottenger and Rudolf Eigenmann. Parallelization in the presence of generalized induction and reduction variables. Technical Report 1396, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1995.

[15] Jay Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[16] 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.