A Context-Sensitive Pointer Analysis Phase in Open64 Compiler

Tianwei Sheng, Wenguang Chen, Weimin Zheng
Department of Computer Science and Technology
Tsinghua University, China
tianwei.sheng@gmail.com, {cwg, zwm-dcs}@tsinghua.edu.cn

Abstract

The precision of the pointer analysis plays an important role in compiler and other software understanding tools. Context-sensitivity is proven to be an effective methods to improve the precision of the final results of pointer analysis. Many context-sensitive pointer analysis methods have been proposed, however, few of them have been incorporated into production compilers. In Open64 compiler, it still uses a flow-insensitive, context-insensitive, and field-insensitive pointer analysis.

In this paper, we design and implement a cloning-based inter-procedural context-sensitive pointer analysis in Open64 compiler. We expect that the new pointer analysis phase will improve the performance and give a good infrastructure for any program analysis tool based on Open64 compiler.

Keywords: Pointer analysis, Context-sensitive, Cloning-based

1 Introduction

Many compiler optimizations and program analysis methods for imperative programming languages rely on the pointer information. Generally, more precise pointer information clients get, more precise results they can produce. For instance, partial redundant elimination (PRE) optimization can hoist the $*q$ out of loop as loop invariant in Figure 1 if we can get precise pointer information that p and q point to different memory location. In Figure 2, a simple static lock-set based race detection tool [1] can easily report that the memory reference for $*q$ at Line 9 is race candidate if we know q may point to the shared variable y.

Pointer analysis methods mainly can be classified into two categories–flow-sensitivity and context-sensitivity[2]. Besides, field-sensitive is also an important property that models the way to handle aggregate types. Flow sensitive pointer analysis is very precise because it respects program's control flow. However, it is also very expensive because it needs to compute points-to solution for every program point. Therefore, very little progress has been made in the flow-sensitive dimension. Context-sensitive analysis considers calling context when analyzing a function. Field-sensitive pointer analysis can distinguish the individual elements of the aggregate variables. Independent of these analysis techniques, there

```
1
2  void foo(int index,int *p, int *q) {
3
4     int i;
5
6     for(i = 0; i < index; i++)
7       {
8
9         *(p + i) = *q;
10
11      }
12
13 }
```

Figure 1: An example in which $*q$ is a loop invariant if we can get p and q point to different memory locations

```
1  pthread_mutex_t lock1;
2  pthread_mutex_t lock2;
3  int x, y;
4
5  void foo(int*p, int *q) {
6    pthread_mutex_lock(&lock1);
7    *p = 1;
8    pthread_mutex_unlock(&lock1);
9    *q = 2;
10 }
11 void bar() {
12   p = &x;
13   q = &y;
14   foo(p,q);
15 }
```

Figure 2: An example in which $*q = 2$ is a race candidate if we can get q points-to shared variable y

are two categories of methods to model the memory locations in the problem ,inclusion-based and unification-based. Inclusion-based [3], also known as Andersen-style method, is that points-to set of pointer on the right side of a pointer assignment is a subset of the points-to set of pointer on the left side. Unification-based[4], also known as Steensgaard-style method, is that points-to sets of the pointer on two sides of an pointer assignment should be unified together. The unification-based algorithm is very fast and has nearly linear time complexity while inclusion-base algorithm is more precise but has cubic complexity. The context-sensitive and flow-sensitive methods can be combined with those model-

ing methods independently. In this paper, we focus on the context-sensitive pointer analysis with the unification method.

In Figure 3, we use an example to briefly explain the above algorithms. For the flow-sensitive methods, different program points have different points-to solutions and q has different points-to solutions with different modeling methods at program point 4. For the flow-insensitive methods, all program points have same points-to solution and q has different points-to solution for inclusion-based and unification methods again. For context-sensitive pointer analysis, the concepts are similar with flow-sensitive, we will describe them later in this paper.

Most of recent studies focus on the context-sensitive pointer analysis because it is proven to be an efficient method to improve precision of the pointer analysis. In Figure 4, a context-sensitive analysis can compute much more precise points-to solution because it can distinguish different calling contexts. Currently, two methods for context-sensitive pointer analysis have been proposed. One is based on the graph-reachability[5], which models the pointer analysis problem into a graph-reachability problem. The other one is based on the cloning that clones the call graph nodes [6] or inlines the callee's pointer information into caller [7] to achieve context sensitivity information. Similar with flow-sensitive analysis, the cloning-based methods can be combined with inclusion-based and unification-based modeling methods.

Chris Lattner et al. [7, 8] have proposed a context-sensitive points-to analysis with heap cloning–Data Structure Analysis (DSA) and implemented it in LLVM compiler. They combined the context-sensitive analysis with the unification-based method and claimed that their algorithm can scale to 355K lines of code. DSA can handle incomplete programs and construct the complete call graph on the fly. They also come up with several engineering choices that overcome bottlenecks of the cloning based methods.

In open64, the current pointer analysis still uses a context-insensitive, field-insensitive algorithm that is imprecise for production use. Our design and implementation for the context-sensitive pointer analysis in Open64 is largely based on the DSA framework in LLVM. However, the intermediate representation (IR) and inter-procedural analysis (IPA) framework of Open64 are different from that in LLVM. In this paper, we present our specific design and engineering tradeoffs in Open64. Overall design mainly includes following items.

- We reorganize the existing points-to analysis phase in Open64 and put a new phase at the beginning of the IPA phase.
- We implement a context-sensitive, field-sensitive, inter-procedural pointer analysis. For structure type, we create different alias nodes for each member, and collapse them when inconsistent access pattern is met. We use the unification method for the case of pointer assignment and cloning between caller and callee.
- For global variables, we use a similar global variable graph to hold all information about global variables in the program. In the algorithm, we update the local graph and global variable graph interactively.
- To verify the results of pointer analysis, we design an inter-procedural mod-ref phase that is treated as a client of the new pointer analysis phase. We plan to implement a simple lock-set based static data race [1] phase to tune the pointer analysis.

The rest of this paper is organized as follows: In Section 2 we first describe background of our work followed by the overview of the new pointer analysis phase. Section 4 presents the detailed design and implementation of the new pointer analysis. The preliminary experimental results are reported in Section 5. Section 6 gives a discussion of related work. We conclude in Section 7.

2 Background

In this section, we first give some definitions that will be used in this paper. Then we present a summary of the IPA framework and existing pointer analysis algorithm in Open64. Finally we describe our new design of the context-sensitive pointer analysis and some conservative tradeoffs.

2.1 Definition

Definition 2.1 (Alias Graph)**.** *An alias graph is the approximation of the alias relation of program variables in a function where each node (alias node) denotes the memory location and the edge denotes the points-to relation between memory locations. Some special nodes, such as return nodes,* ***Callsite*** *nodes are recorded to do inter-procedural analysis. Each member of an structure has its own alias node that is linked together. The alias node for the structure has a link to the alias node of first member.*

Figure 5(b) is the alias graph for the example in Figure 5(a). The **Callsite** node contains return value, called function and actual parameters. A **return** node is recorded. In this example, since the **Callsite** return a pointer to a STR node, we create a dummy structure alias node that is pointed to by the r field of the **Callsite** node. This structure alias node is incomplete and will be resolved after inlining bar's alias graph into foo at the bottom-up phase.

Definition 2.2 (Unification)**.** *When handling pointer assignment, the points-to sets of two pointers on two sides of the pointer assignment are unified together so that they have the same points-to sets*

In Figure 6, p and q are unified into one points-to set, s and t point to the the unified points-to set.

Definition 2.3 (Alias Graph Cloning)**.** *Alias graph cloning means that reachable alias node in called function is cloned into calling function. The reachable alias nodes come from the actual-formal parameters binding, return values, global variables.*

```
int *p,*q;
int x,y,z;
void foo(int i)
{
 q = &z;
 if(i)
  {
    p = &x;//program point 1
  }
 else{
   p = &y; //program point 2
 }
 //program point 3
 p = q;
 //program point 4
}
```

(a) example code

	Flow-sensitive		Flow-insensitive	
	Inclusion-based	unification-based	inclusion-based	unification-based
program point 1	p->{x} q->{z}	p->{x} q->{z}	p->{x,y,z} q->{z}	p->{x,y,z} q->{x,y,z}
program point 2	p->{y} q->{z}	p->{y} q->{z}	same as point 1	same as point 1
program point 3	p->{x,y} q->{z}	p->{x,y} q->{z}	same as point 1	same as point 1
program point 4	p->{x,y,z} q->{z}	p->{x,y,z} q->{x,y,z}	same as point 1	same as point 1

(b) points-to solution for different methods

Figure 3: An example to explain different algorithms. For flow-sensitive algorithm, for the assignment $p = q$, we assume there are no kill update (the most precise solution is that p only points to z), and use inclusion-based and unification-based methods to compute the solution.

```
int foo(int **p, int*q) {
 *p = q;
 *q = 1;
}
int main()
{
  int x, y;
  int *s, *t;

  foo(&s,&x);
  foo(&t,&y);
  printf("x = %d\n",x);
  printf("y = %d\n",y);
  return 0;
}
```

(a) example code

```
Context-sensitive:
p1 = &s; p2 = &t;
q1 = &x; q2 = &y;
*p1 = q1 *p2 = q2
*q1 = 1; *q2 = 1;
Solution: s->{x}, t->{y}

Context-insensitive:
p = &s; p = &t;
q = &x; q = &y;
*p = q
*q = 1
Solution: s->{x,y}, t->{x,y}
```

(b) points-to solution

Figure 4: A simple example to illustrate the context-sensitive pointer analysis

2.2 Inter-procedural Analysis Framework of Open64 Compiler

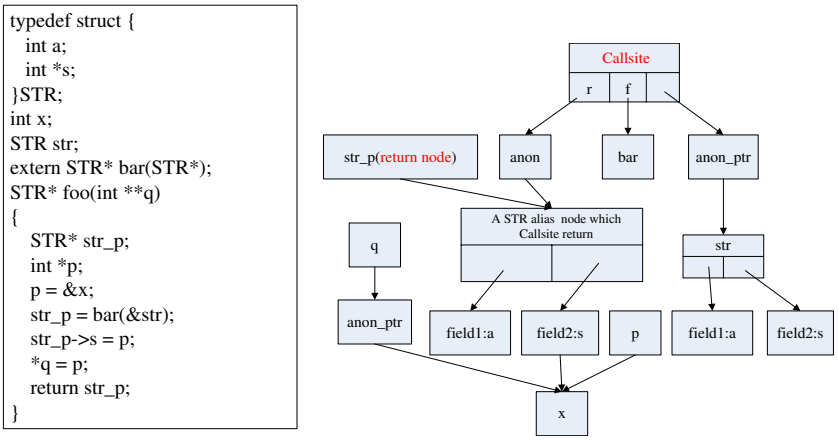
Figure 7 gives inter-procedural analysis framework of Open64 compiler. IPL (inter-procedural local phase) is responsible for collecting the summary information for all functions. IPA (inter-procedural analysis) make use of the summary information from the IPL and applies different analysis. The IPA does not check the IR and only work on the summary information. Typical analysis phases include inter-procedural constant propagation, inter-procedural inline analysis, etc. IPO (inter-procedural optimization) reads the IR and transforms the code according to the decisions of the IPA.

2.3 Existing Pointer Analysis Phase in Open64

The existing pointer analysis in Open64 is still based on the Steensgaard algorithm [4] that is context-insensitive, field-insensitive. Open64 implemented both intra-procedural and inter-procedural version algorithms. The inter-procedural pointer analysis is designed only for intra-procedural optimization phase, and located at the end of IPA phase. In the intra-procedural phase, combined with other memory disambiguation techniques that are similar with the work in [9], the pointer analysis can answer most of the alias queries from client optimization phases.

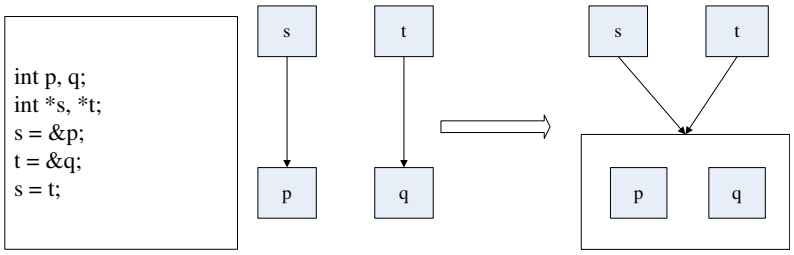
Overall, the whole existing alias analysis framework of Open64 is presented in Figure 8.

There are several shortcomings in the existing framework. First, the IPA cannot make use of any pointer information because the pointer analysis is at the end of the IPO phase. The inter-procedural pointer analysis is only designed for intra-procedural optimization phase. Second, the pointer analysis is still context-insensitive and field-insensitive, which is fairly imprecise. Actually, from our experimental results, most of the alias queries are resolved by the Base-offset-size rule that comes from symbol table information and SSA-based flow



(a) example code (b) Alias graph

Figure 5: Basic constructs of an alias graph. $anon\_ptr$ and $anon$ are the temporal alias nodes.



(a) example code (b) unification graph

Figure 6: An unification example

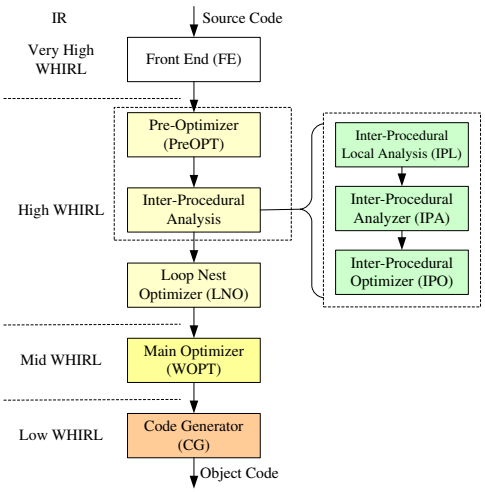


Figure 7: The inter-procedural analysis framework in Open64

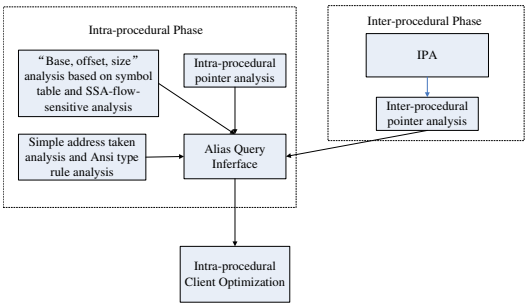


Figure 8: The existing alias analysis framework in Open64

sensitive analysis. On the other hand, the compiler has to do conservative analysis for parameter passing and procedure calls even with an inter-procedural pointer analysis.

3 Overview of The New Pointer Analysis Phase

For the new context-sensitive pointer analysis, we use a standard inter-procedural data flow analysis to compute the pointer information for the whole program. The main algorithm is divided into three steps:

- **Local Phase**

  For this phase, there are several design tradeoffs for Open64. In the framework of Open64, the IPA will only work on summary information from the IPL and will not check the IR because of the scalability problem. The

only phase that traverses the IR of whole program is at the IPO phase. For the new pointer analysis, because we need to collect every pointer related statements, we create a separated phase at the beginning of the IPA phase instead of collecting summary information at the IPL and passing to the IPA. This local phase traverse every PU (Program Unit) and create the local alias graph for them. For any global variable that has occurrence in the local graph, we also record it in the global variable graph. For the call sites, as in [7], we create a **Callsite** node that incorporates the parameters and return values information.

We recognize standard library calls in this phase and summarize their side effects by hard code in the compiler. The final local graph do not contain any call sites for these library calls.

The result local graphs only contain intra-procedural pointer information and the **Local** phase is also the only phase that checks the real IR. The following two phases work on the the local graph and eliminate the incomplete information through graph inlining.

We put the local phase after the DFE(dead function elimination) phase. DFE can use some simple address-taken analysis techniques to eliminate some unreachable functions. Furthermore, if some dead functions reference global variables, it also can improve the precision.

- **Bottom-up Phase**

  The bottom-up phase will eliminate the side effects of call sites through inlining the pointer information of callee into caller. We sort the call graph nodes in a topological order, and then visit these call graph nodes in post topological order. During the visiting, we use a standard strong connected component (SCC) detection algorithm to find SCCs. For a call graph node, we first incorporate the global variables information and copy any reachable alias nodes into the local graph. Then we inline the reachable nodes in the callee to caller through actual and formal parameters binding. The bottom-up phase do not copy non-reachable alias nodes into the alias graph of caller. Finally, we update the global variable graph according to the new local alias graph.

  For any indirect call that is introduced by function pointer, we copy the **Callsite** node into caller if the pointed function has not been resolved yet.

  When all information has been propagated into main function, all the local graphs do not have any incomplete information introduced by direct function calls.

- **Top-down Phase**

  Even after the bottom-up phase, for a function, we still cannot know the information of formal parameters and have to do conservative analysis for these parameters. In order to eliminate these incomplete information, for a function, we copy nodes in all of its callers that are reachable from the formal parameter into this function.The final Top-down alias graphs contain all the results of our pointer analysis algorithm.

```
1  void foo(int *p,int *q) {
2     int i;
3     for(i = 0; i < 10; i++)
4        *(p + i) = *q;
5  }
6  void bar() {
7    int a[10],x;
8    x = 1;
9    foo(a,&x);
10 }
```

Figure 9: A simple example to explain the necessity of top-down phase

  In Figure 9, the leaf function foo still can not know whether p and q point to different memory locations after bottom-up phase. However, after top-down phase, the a and x are copied into foo so that they are pointed to by p and q respectively.

The alias graphs for all functions are attached into their call graph nodes and have alias query interfaces. For scalar variables, the client can provide symbol table index (ST) of two variables and invoke the alias query interface, the alias graph will return the results if they are aliased or not. For indirect reference , the client first locates the alias node for them through access path ($**p$,$***q$,etc), then determines whether these two alias node in the same alias set or not.

Now the IPA and BE are separated dynamic libraries and any information propagating between them must be stored into the intermediate files. Since we use the unification method, we are planning to use a cheap method to pass these alias information to intra-procedural phase instead of passing those alias graphs one by one.

4 Design and Implementation

In this section, we describe the detailed design and implementation of the new context-sensitive pointer analysis. First we present how we implement the unification and the cloning method to get context-sensitive pointer information. Then we discuss the design of field-sensitive analysis and global variable graph. Finally we briefly describe the mechanism to handle function pointer and recursion.

4.1 Context-sensitive Analysis

As discussed in previous sections, we use cloning-based method to achieve context-sensitivity. It clones the pointer information of callee into caller. For the example in Figure 10, foo is invoked twice and passes different parameters.

At the **Local** phase, the algorithm collects pointer information and create the local alias graphs for all functions. For this example, 11(a) and 11(b) are the local alias graph for the

```
1  int foo(int **p, int*q)
2  {
3    *p = q;
4    *q = 1;
5  }
6  int main()
7  {
8    int x, y;
9    int *s, *t;
10
11   foo(&s,&x);
12   foo(&t,&y);
13   printf("x = %d\n",x);
14   printf("y = %d\n",y);
15   return 0;
16 }
```

Figure 10: Another simple example to illustrate the context-sensitive pointer analysis

functions foo and $main$. Any variable that can be reachable from parameters of function call must be treated conservatively since they may be modified by other functions.

At the bottom-up phase, since the call graph has been constructed and all actual-formal parameter information bindings have been incorporated into the call graph nodes, the pointer information of foo is cloned into main. Finally we get the s points-to x and t points-to y in main function. After cloning a callee, the corresponding **Callsite** node in the caller's alias graph is deleted.

The bottom-up phase is the main phase where the context-sensitive pointer analysis occurs. The top-down phase sometime cannot give us much context-sensitivity since our method is flow-insensitive.

At the top-down phase, the pointer information of main will propagate down to foo. We do not get much precise information for foo because the p still points-to s and t while q points-to x and y. This is because we do not compute flow-sensitive nor path-sensitive information. However, if we only care about the pointer information in $main$ function, the alias graph after bottom-up contains all the pointer information.

4.2 Field-sensitive Analysis

Field-sensitive[10] means that every structure member has its own distinguished memory location. Our algorithm models this property through creating an alias node for each member of structure variables. The structural alias node is labeled as **Collapsed** if there are inconsistent type casts over this alias node.

For the example code in Figure 12, the alias graph is presented in Figure 13. In this graph, all aggregate variables have a link to its first member. Every node has the information about offset, size, field-id. Given a field-id/offset and an structure node, we can find the corresponding sub-node in this aggregate node.

The IR of line 25 and 26 are listed in Figure 14. For line 25, to do field-sensitive analysis, from the second kid of

```
1  typedef struct{
2    int * p;
3    int f;
4    float g;
5  }INNER;
6  typedef struct {
7    int a;
8    INNER w;
9    float b;
10 }MIDD;
11 typedef struct{
12   int c;
13   MIDD d;
14   INNER m;
15   float e;
16   int *x;
17   int v;
18 }OUTER;
19
20 OUTER out;
21 int main()
22 {
23   int y;
24   OUTER *p = &out;
25   p->x = &y;
26   int ** q = &(p->x);
27   **q = 4;
28   printf("%d\n", *(p->x));
29 }
```

Figure 12: Field-sensitive example

```
1  LOC 1 31   p->x = &y;
2  U8LDA 0 <2,1,anon1> T<50,anon_ptr.,8>
3  U8U8LDID 0 <2,2,anon2> T<48,anon_ptr.,8>
4  U8ISTORE 64 T<48,anon_ptr.,8> <field_id:14>
5  LOC 1 32   int ** q = &(p->x);
6    U8U8LDID 0 <2,2,anon2> T<48,anon_ptr.,8>
7    U8INTCONST 64 (0x40)
8  U8ADD
9  U8STID 0 <2,3,anon3> T<51,anon_ptr.,8>
```

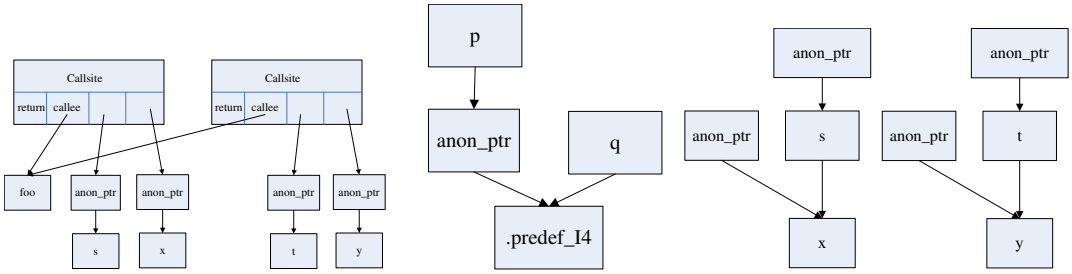
Figure 14: The IR of line 25 and 26 in Figure 12

$ISTORE$, we get the alias node for p. Since it is $ISTORE$ and the offset, field-id are not zero, we get the alias node for the member of the structure node. Here it is the alias node for member x. Then we add an edge to y from the alias node for member x. For the line 26, it is a pointer arithmetic. We add a refining phase to handle such simple pointer arithmetic and get a temporal node that point to the member x. The $STID$ statement will unify the points-to set of q and temporal node and make q point to the member x of the structure node.

In open64, we can not simply distinguish patterns between the start address of a nested structure and the address of the first member of this nested structure. Now we solve this problem through matching the type information of STID and field node. To do this, we add an link from the first member to the structure node itself. If the type of field node for an ADD expression does not match the type of STID, we return the structure alias node.

4.3 Global Variable Graph

During the bottom-up phase, we need to inline any reachable node to caller from callee. In this process, we also have to copy all global variables since we do not know if they will be accessed or not. In order to ensure correctness, we copy all global variables into main in the bottom-up phase and propagate down to all functions. This process incurs unacceptable overhead to the whole algorithm. To overcome such problem, we use a similar method [7, 11] to create a separate global variable graph to hold those global variables information and do interaction between global variable graph and local alias graph.



(a) Local graph for main function (b) Local graph for foo function (c) Alias graph for main after cloning

Figure 11: The local and bottom-up alias graph for the code in Figure 10

```
1  int *p,  *q,  **s;
2  int x, y;
3  void foo_1() {
4    s = &p;
5  }
6  void foo_2() {
7    p = &x;
8  }
9
10 void foo_3() {
11   q = &y;
12 }
13
14
```

```
14 void bar(int **m, int *n)
15 {
16   *m = n;
17 }
18  void bar_1() {
19   bar(&p,&y);
20 }
21 int main()
22 {
23   foo_1();
24   foo_2();
25   foo_3();
26   bar_1();
27 }
```

Figure 15: The example code to illustrate the usage of global variable graph

For the example in the Figure 15, at the bottom-up phase, when we are cloning the function bar into bar_1, we do not know the pointer information about the global variable pointer p. At that time, before cloning, we first need to update the local graph according to the global variable graph where p points-to x. After cloning, p points-to x and y, we need to update the global variable graph again and then other functions can know this updated information for these global variables.

4.4 Function Pointer And Recursion

For function pointer, we treat it as a normal pointer during the analysis. When the function pointer is invoked in the program where there is an indirect call, we create a **Callsite** node and the f field points-to the alias node for the function pointer. During the bottom-up phase, if the callee still has indirect calls, these **Callsites** need be copied to the caller. In some cases, the function pointer may be resolved after cloning, then the SCCs algorithm will revisit the caller and new resolved callee again.

To handle the recursion, we merge the alias graphs of the functions in the same SCC into one alias graph and sacrifice the context-sensitivity within the same SCC as the way in [7]. After visiting the SCC, the functions inside the SCC may become the callee or the caller of functions which are out of the SCC. In order to provide the right formal parameters and return value information when visit such callsites, we create a mapping between these information and IPA_NODE for each function in the merged SCC graph.

4.5 Some Tradeoffs

We work on the High-level WHIRL (Winning Hierarchical Intermediate Representation Language) that still preserves the ARRAY operator. In our algorithm, we treat the whole array as an element and will not distinguish individual elements. For pointer arithmetic, we add a refining phase to identify simple pointer arithmetic statement. For those complicated pointer arithmetic, we make them point to an unknown node.

5 Preliminary Experimental Results

We are still implementing and testing the new pointer analysis phase. In this section we only give a preliminary results for our current work. Now the new phase can get the final alias results for medium-sized program, such as SPEC 2000. However, it still cannot pass large programs at bottom-up phase, such as mysql. We give our preliminary result in 1. The mysql is built with the *-all-static* flag and all sub-components will be compiled into static libraries(the .a libraries contain object files that are in WHIRL format). This gives us much more large inputs and reduces the number of extern library calls. In order to verify the overhead of field-sensitive analysis, we add a flag to disable the field-sensitive analysis.

From the preliminary data, we can get the following insights:

- The Bottom-up phase is unlikely to increase the number of alias nodes exponentially. This is because the cloning phase only clones reachable nodes to the caller from the callee. Actually, most of the cloning phases only add the new points-to relations which are introduced by the function call and will not copy any node. These data is consistent with the fact that the scope of local variables in the callee does not include the code in the caller.
- For some programs, such as 177.mesa and mysql, the field-sensitive analysis incurs very much overhead since the algorithm creates an alias node for each member of the structure variables. The field-insensitive will have

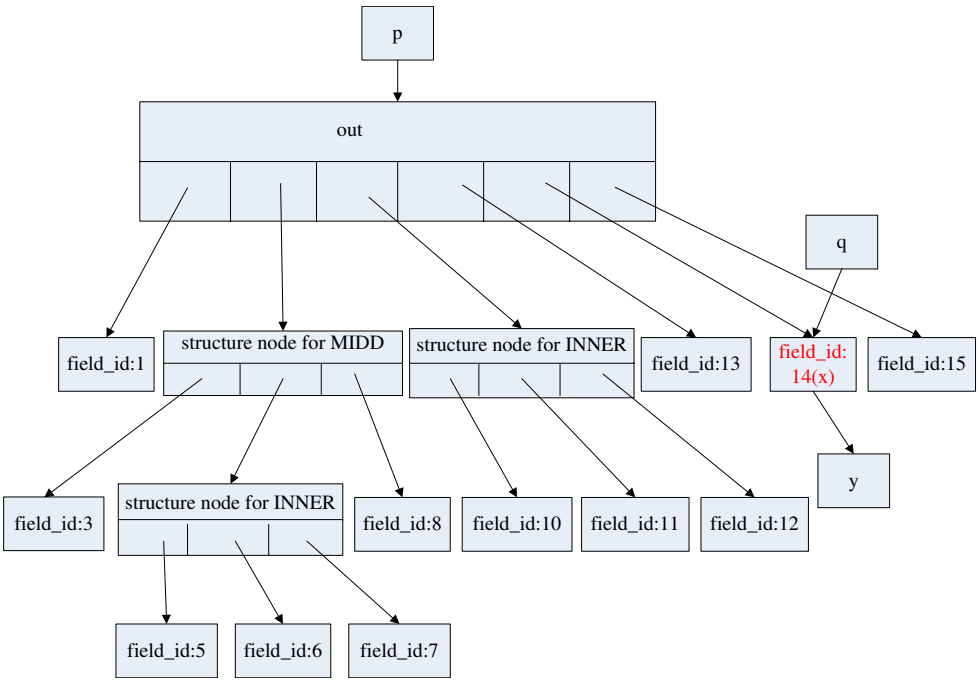


Figure 13: The alias graph for the main function in Figure 12. Note that in implementation, the field members are linked together and the structure node has a link to the first field member. Furthermore, in the IR definition of Open64, the nest structure is flatten and the nest structure itself occupies an field_id number. That's why the first member of MIDD has the field_id 3 and not 2.

much less number of alias nodes. We also add a field number limit which can disable the field-sensitive for the variables that have large numbers of fields.

Now our work is still very preliminary and can not pass large program at the bottom-up phase. Furthermore, we are implementing some client optimizations phases, such as mod-ref analysis, inter-procedural lock-set based data race detection, to evaluate the precise of the new context-sensitive analysis compared with the existing algorithm in Open64.

6 Related Work

Pointer analysis have been studied for a long time. A lot of methods have been proposed that focus on the tradeoff between precision and scalability. Recently, two dimensions are active, one is to improve the scalability of inclusion-based context-insensitive method and the other one is the context-sensitive analysis combined with unification method.

For the inclusion-based context-insensitive method, Ben et al. [12] proposed lazy cycle detection and hybrid cycle detection that reduce both the time and memory overhead of the algorithm significantly. Their method has been incorporated into production compilers, such as gcc and LLVM.

For the context-sensitive direction, studies prove that it is hard to scale without the combination with unification. Wilson et al. [13] use a partial transfer function to summarize the pointer information of a function and instantiate the callee at different call sites. Compared with the cloning based method, they cannot get context sensitivity for heap objects. Besides, their methods only can compile medium-sized program. Nystrom,et al[14] also proposed a summary-based context-sensitive pointer analysis method and use bottom-up,top-down traverse over the call graph. Liang et al. [11] proposed a MOPPA method to compute context-sensitive pointer information. They also use a global variable graph to hold all global variable information. All these methods have not been tried in the production compiler. So it is hard to evaluate if they can scale to large programs.

Our work is largely based on the DSA framework [7]. They used three-steps inter-procedural analysis and graph inlining to get context-sensitivity. In addition, they applied a lot of engineering methods that make their algorithm very practical. They implemented the algorithm in LLVM compiler that is very convincing for their experimental results.

7 Conclusion and Future Work

In this paper, we design a context-sensitive pointer analysis phase in Open64. The new phase can compute more precise pointer information than existing alias analysis method in Open64. Furthermore, the inter-procedural analysis can benefit from the new pointer analysis due to the new phase ordering. There are a lot of future work. We will continue to test the new phase and get complete experimental results . We also plan to implement an inter-procedural data race detection phase that makes use of the results of the new pointer analysis

Table 1: The total alias node number after Local phase and Bottom-up phase. FS is Field-Sensitive and FI is Field-Insensitive. U means that current implementation can not pass the benchmark.

Benchmark	Number of Functions	max SCC	Local Phase		Bottom-up Phase	
			FI	FS	FI	FS
164.gzip	106	0	4459	4719	4622	4863
186.crafty	109	2	13710	15928	14011	16373
254.gap	854	20	49450	67339	72182	91062
256.bizp2	74	0	2493	2250	2584	2278
175.vpr	272	0	10413	12317	10706	13671
181.mcf	26	0	718	2259	732	2636
197.parser	324	3	9255	13016	9565	14166
253.perlbmk	1076	322	48084	72215	67870	93364
255.vortex	923	38	46596	76219	81961	117293
300.twolf	191	0	13564	14853	13758	16236
177.mesa	1106	0	36226	719659	60706	1157327
179.art	26	0	931	912	943	1030
183.equake	27	0	1223	1278	1236	1389
188.ammp	179	2	8196	23618	8407	24286
mysql	52875	U	743318	7216837	U	U

phase. Since the static data race detection relies on the pointer information very much, we are planning to learn the interaction between them. Finally we are designing consistent alias query interface that makes it easy to compare different pointer analysis algorithms.

References

[1] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-811-4. doi: http://doi.acm.org/10.1145/1287624.1287654.

[2] Michael Hind. analysis: Havent we solved this problem yet. In *In PASTE*, pages 54–61. ACM Press, 2001.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[4] B. Steensgaard. Points-to analysis in almost linear time, 1996. URL `http://citeseer.ist.psu.edu/steensgaard96pointsto.html`.

[5] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 387–400, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1134027.

[6] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. doi: http://doi.acm.org/10.1145/996841.996859.

[7] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *SIGPLAN Not.*, 42(6):278–289, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1273442.1250766.

[8] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. *See* `http://llvm.cs.uiuc.edu`.

[9] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for c programs. *SIGPLAN Not.*, 36(5):47–58, 2001. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/381694.378806.

[10] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for c. In *In ACM workshop on Program Analysis for Software Tools and Engineering (PASTE*, pages 37–42. ACM Press, 2004.

[11] Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *In Static Analysis Symposium*, pages 279–298. Springer-Verlag, 2001.

[12] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 290–299, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: http://doi.acm.org/10.1145/1250734.1250767.

[13] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2. doi: http://doi.acm.org/10.1145/207110.207111.

[14] Erik M. Nystrom, Hong-Seok Kim, and Wen-Mei W. Hwu. *Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis*. 2004. URL `http://www.springerlink.com/content/59q3n3tkmmecg8qv`.