To appear in Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing, Columbus, Ohio, August 10–12, 1995. To be published by Springer–Verlag.

Connection Analysis: A Practical Interprocedural Heap Analysis for C *

Rakesh Ghiya and Laurie J. Hendren

School of Computer Science, McGill University
Montréal, CANADA
{ghiya,hendren}@cs.mcgill.ca

Abstract. This paper presents a practical heap analysis technique, *connection analysis*, that can be used to disambiguate heap accesses in C programs. The technique is designed for analysing programs that allocate many disjoint objects in the heap such as dynamically-allocated arrays in scientific programs.

The method statically estimates connection matrices which encode the connection relationships between all heap-directed pointers at each program point. The results of the analysis can be used by parallelizing compilers to determine when two heap-allocated objects are guaranteed to be disjoint, and thus can be used to improve array dependence and interference analysis.

The method has been implemented as a context-sensitive interprocedural analysis in the McCAT optimizing/parallelizing C compiler. Experimental results are given to compare the accuracy of connection analysis versus a conservative estimate based on points-to analysis.

1 Introduction and Background

Optimizing and parallelizing compilers rely upon accurate static disambiguation of memory references i.e. determining at compile-time, if two given memory references always access disjoint memory locations. Although there has been a long history of developing methods for disambiguating array references, there has been an increasing interest in a variety of methods for disambiguating pointer references. This is becoming more important as optimizing and parallelizing compilers are being developed for languages supporting pointers such as C and FORTRAN90.

The pointer analysis problem can be divided into 2 distinct subproblems: (1) disambiguating pointers that point to objects on the stack, and (2) disambiguating pointers that point to objects on the heap. There has been a considerable amount of work in both of these areas [JM81, JM82, LH88, Lar91, Gua88, HPR89, Har89, CWZ90, HN90, LR92, CBC93, Deu92, Deu94, PCK93, EGH94, WL95, Ruf95], although more attention has been paid to actually implementing methods that work well for stack-allocated objects [LR92, CBC93, EGH94, Ruf95, WL95]. A complete discussion and comparison of these methods can be found in [Ghi95].

Stack-directed pointers exhibit the important property that their targets always possess a name (the name of the variable allocated to that location on the stack). Using this property, pointer relationships can be conveniently captured in the form of *points-to* pairs where the points-to pair `(p,x)` denotes that pointer variable `p` points to the data object `x`.

* This work supported by NSERC, FCAR, and the EPPP project (financed by Industry Canada, Alex Parallel Computers, Digital Equipment Canada, IBM Canada and the Centre de recherche informatique de Montréal).

Unfortunately this nice property of having a static name for all targets does not hold for heap-allocated data items. In fact, all the objects in the heap are allocated via a memory allocation function and are *anonymous*. Heap objects cannot be referenced by their name, they can only be accessed through pointer dereferences like `*r`, `r->item` and `a[i]`, where `r` and `a` are heap-directed pointers. One might imagine that one could generate symbolic names for heap objects, but this is also difficult as a potentially infinite number of them can be created. To further complicate the problem, objects in the heap are dynamically linked, and more importantly delinked. Hence, there is no natural way of naming even collections of objects (e.g. linked structures). Unlike arrays, both the number of linked structures and the number of objects belonging to a given linked structure, vary dynamically. Thus, in order to estimate more accurate information about heap-directed pointers, a different approach is required.

In addition to designing the heap analysis itself, it is also important to determine how the heap analysis interacts with the stack analysis, and to design an analysis that can be effectively implemented in real C or FORTRAN90 compilers. It is interesting to examine three recently implemented strategies. Landi and Ryder have reported on an implementation of an interprocedural strategy for C that estimates alias information in terms of pairs of *object names* [LR92]. An object name consists of a variable and a (possibly empty) sequence of dereferences and field accesses. Typical alias pairs are: `(**a, *b)`, `( *(a->next), *(b->next) )`. In the presence of recursive data structures, the number of object names is infinite. To avoid this, they k-limit object names. Choi, Burke and Carini [CBC93] have been implementing a method for FORTAN90, and they also compute aliases of pairs of *access paths*. Their access paths are similar to object names [LR92]. However, they do not use access paths to name heap objects. Instead, they use the place (statement) in the program, where an anonymous heap object is created, to name it, as in [CWZ90]. To avoid giving the same name to heap objects created at the same statement, but along different call-chains, they qualify the names with procedure strings. A recent approach for C is the implementation of a context-sensitive method for the SUIF compiler system [WL95]. In this approach, a points-to representation is used, however, they use the concept of *location sets* to specify both a block of memory, and a set of positions within that block. Blocks of memory that are heap-allocated are labeled by their allocation context.

In all three of these approaches the stack-directed and heap-directed pointer problems are solved together. In contrast, our approach is to decouple the problems and to first solve the stack-directed pointer problem using points-to analysis [EGH94, Ema93], and then use the result of points-to analysis as a starting point to solve the heap-directed pointer problem. The motivation for decoupling the problems is that the solutions for the two problems are quite different, and by concentrating on each problem separately we can design appropriate abstractions and analysis rules. For the stack-directed pointer problem we calculate direct pointer relationships between *named* locations, whereas for the heap-directed pointer problem we need to collect more general relationships (such as which heap-directed pointers possibly lead to a common node). This decoupling is also beneficial from a software development point of view. By using a simple model for the heap in points-to analysis, we can simplify the points-to analysis rules, and reduce the number of objects that must be abstracted (we have only one object called *heap*, whereas the other combined approaches must use many names for objects in the heap). This leads to a faster and more space-efficient points-to analysis. Similarly, our heap analysis is simplified and faster because we can use the result of points-to analysis to locate only those pointers that point to the heap, and then run the heap analysis using this subset of pointers.

In fact, we have a hierarchy of analyses that abstract *relationships* between heap-directed pointers. As one goes up the hierarchy, a more precise solution is obtained, but at the cost of a more complex and expensive analysis. The *level-0* analysis is simply the points-to analysis that treats the entire heap as one named location. For programs with few heap objects, this is enough. This paper focuses on the *level-1* heap analysis: *connection analysis.* Connection analysis is targeted towards programs that allocate a number of disjoint data structures in the heap. Scientific applications written in C typically exhibit this feature, as they use a number of disjoint dynamically allocated arrays. Connection analysis can also be used to distinguish between different linked data structures. The analysis was designed to provide a simple, but useful, analysis that would provide accurate results for its intended domain of applications. The rest of the paper is organized as follows. In Section 2 we introduce the analysis and give some high-level analysis rules assuming a simple model where stack-directed pointers and heap-directed pointers are clearly separated. The method has been fully implemented in the McCAT compiler as a context-sensitive interprocedural analysis. In Section 3 we give a brief overview of our implementation of this method and we discuss the most pertinent features. We present some empirical data in Section 4, to demonstrate the cost and effectiveness of this abstraction for its intended domain of applications. Section 5 gives our conclusions and briefly describes our future work.

2 Connection Analysis

Connection analysis uses a simple, *storeless* [Deu92], abstraction designed to disambiguate heap accesses at a coarse level, but in an efficient and cost-effective manner. For each program point the analysis computes a *connection matrix*, which is a boolean matrix summarizing the connectivity of heap objects. A *heap object* is defined as an object allocated in the heap memory. Our connection analysis is performed with respect to a connection matrix abstraction that is designed to disambiguate heap accesses at the data structure level. The term *data structure* in this context represents a connected region in the heap i.e. if the heap is viewed as an undirected graph with heap objects as nodes and links between them as edges, each connected component forms a separate data structure. Two disjoint data structures contain no common heap objects. For example in Figure 1, the heap consists of two data structures: one pointed to by pointers `p` and `q` and the other pointed to by pointers `r`, `s` and `t`. Note that we cannot give names to these data structures, we can only refer to them as being pointed to by a given set of pointers.

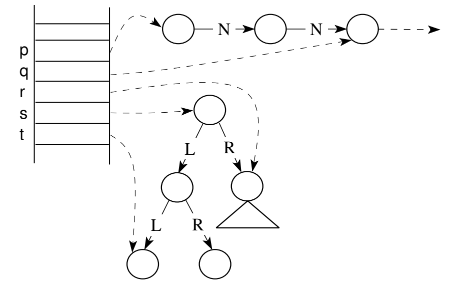
With the above definitions, given any two heap-directed pointers say `p` and `q`, connection matrix abstracts the following *program-point-specific* relationships:

C[p,q] = 1: Pointers `p` and `q` *possibly* point to heap objects belonging to the same data structure. In our terminology, pointers `p` and `q` are considered to be *connected*, or to have a *connection* relationship.

C[p,q] = 0: The heap objects pointed to by pointers `p` and `q` *definitely* belong to different data structures. In other words, pointers `p` and `q` are *not* connected.

The useful information is the negative information. If pointers `p` and `q` are not connected, then heap accesses originating from them will always lead to disjoint heap locations, and thus not interfere. It is *safe* to report two heap-directed pointers to be connected, when they are not. However, if they can point to the same data structure, they should always be reported to be connected.

We illustrate the abstraction in Figure 1. Part (a) shows the structure of heap at a program point, while part (b) shows its abstraction as a connection matrix. The zero in entry C[p,r] indicates that pointers `p` and `r` point to disjoint data structures in the heap. The one in the entry C[s,r] indicates that `s` and `r` point to objects belonging to the same data structure. Note that the entry C[r,t] is set to one, despite the fact that pointers `r` and `t` point to disjoint subpieces of the same data structure. This is because connection matrix is designed to disambiguate heap accesses at the data structure level. More sophisticated abstractions, which can distinguish between subpieces of a data structure itself, are defined in higher levels in the hierarchy of heap analyses.



(a) Stack and Heap

	p	q	r	s	t
p	1	1	0	0	0
q	1	1	0	0	0
r	0	0	1	1	1
s	0	0	1	1	1
t	0	0	1	1	1

(b) Connection Matrix

Fig. 1. An example Connection Matrix

The following are some other important characteristics of the connection matrix abstraction:

- It abstracts relationships only between *stack-resident* heap-directed pointers. As all heap accesses originate from these pointers, their relationships effectively capture the structure of the heap. For example in Figure 1(b), the information that pointers `p` and `s` point to disjoint data structures also simultaneously implies that pointers `p->N` and `s->L` point to disjoint structures.
- For each function in the program, the connection matrix abstracts relationships between all stack-resident pointers which can be heap-directed at some point in the program and are directly or indirectly (through an indirect reference)

accessible from the function. Names are naturally available from the program, for directly-accessible pointers. For indirectly-accessible pointers, special symbolic names are generated by points-to analysis and these names are reused by connection matrix analysis. To know which pointers ever point to heap, the existing points-to information is used. If `p` points-to the heap, then the entry *(p,heap)* will appear in the points-to set.

- If a pointer, say `p`, does not point to a heap location at a given program point, the connection matrix entry C[p,p] is set to zero at that program point. In this case the pointer points to `NULL` or to a stack location.
- The connection matrix relationship is symmetric i.e. for any two heap-directed pointers say `p` and `q`, we always have C[p,q] = C[q,p]. The connection relationships shown in Figure 1(b) illustrate this property. It is used in the actual implementation to reduce the storage requirement by half.

The complete list of the basic statements that can affect heap relationships is given in Figure 2(a). Variables `p` and `q` and the field `f` are of pointer type, variable `k` is of integer type, and `op` denotes the `+` and `-` operations. In this section we give the analysis rules for these eight *basic heap* statements with the restriction that pointers `p` and `q` can only point to heap objects. These rules are simple to describe and clearly illustrate the basic principles of connection analysis. In the next section we discuss the extensions that must be made to handle complete C programs where the effect of stack-based points-to relationships must also be taken into account. The overall structure of the analysis is shown in Figure 2(b). We have the connection matrix C at program point `x` before the given statment, and we wish to compute the connection matrix C_n at program point `y`. To this end, we define an analysis rule for each of the eight statements shown in Figure 2(a). Each rule computes the following sets of relationships:

kill_set: Set of connection relationships killed by the given statement i.e. the set of relationships which were valid before the statment (program point `x`), but are not valid after processing it (program point `y`). The entries corresponding to these relationships should be set to zero in the connection matrix C_n.

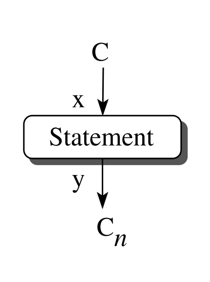
gen_set: Set of connection relationships generated by the given statement. The entries corresponding to these relationships should be set to one in the new matrix C_n.

Let H be the set of pointers whose relationships are abstracted by the connection matrix C. Let `p`, `q`, `r` and `s` represent pointers in this set. Assume that pointers can only point to heap objects or to `NULL`. Further, assume that updating an entry C[p,q] also implies identically updating the entry C[q,p]. This assumption is required due to the symmetric nature of connection relationships. The new connection matrix C_n is computed as shown in Figure 2(c). First, the old connection matrix C is copied over to C_n. Next, the entries in the kill_set are set to zero in the matrix C_n. Finally, the entries in the gen_set are set to one in the matrix C_n, to get the complete new connection matrix.

We now present the analysis rules for the eight basic statements. For each statement, we give the rules for computing their kill and gen sets. The new connection matrix can then be computed as shown in Figure 2(c).

```
1. p = malloc();
2. p = q;
3. p = q->f;
4. p = &(q->f);
5. p = q op k;
6. p = NULL;
7. p->f = q;
8. p->f = NULL;
```

(a)



(b)

Build the new Connection Matrix
$\forall$ r,s $\in$ H, C_n[r,s] = C[r,s]

Delete killed relationships
$\forall$ entries C(r,s) $\in$ kill_set, C_n[r,s] = 0

Add generated relationships
$\forall$ entries C(r,s) $\in$ gen_set, C_n[r,s] = 1

(c)

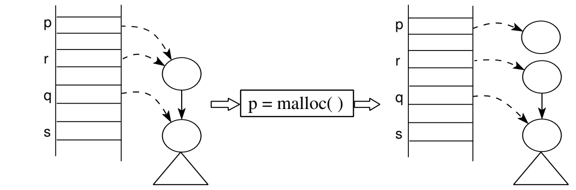
Fig. 2. Computing Connection Matrix C_n from C for Basic Statements

2.1 Allocating new heap cells

`p = malloc()`: In this case pointer `p` points to a newly allocated heap object. All the existing connection relationships of `p` get killed. Also as no other pointer can point to this object, `p` will only have connection relationship with itself. This is stated with the following rule.

$$\text{kill\_set} = \{ \ C(p,s) \mid s \in H \land C[p,s] \ \}$$
$$\text{gen\_set} = \{ \ C(p,p) \ \}$$

The rule is illustrated with the following example. Note that after executing the `p = malloc()` statement, `p` is only connected with itself.



kill_set = { C(p,p), C(p,r), C(p,q) }

gen_set = { C(p,p) }

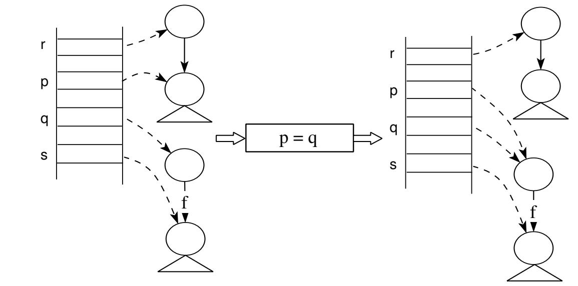
2.2 Pointer assignments

Basic heap statements 2 through 5 in Figure 2(a) (`p = q`, `p = q->f`, `p = &(q->f)` and `p = NULL`), have a common attribute: all of them update the stack-resident pointer `p`, and make it point to a new data structure. They do not modify the structure of the heap itself. Their effect on connection matrix information can be summarized using a general rule, as discussed below.

`p = q` : Pointer `p` now points to the same heap object as `q`, and hence to the same data structure as `q`. All the existing relationships of `p` get killed, and `p` gets connected to all pointers connected to `q`. If `q` is presently heap-directed (C[q,q] = 1), then `p` is also heap-directed after the statement. So the entry C(p,p) is added to the gen_set, if we have C[q,q] = 1. We present the overall rule for this statement below.

$$\text{kill\_set} = \{ \ C(p,s) \mid s \in H \land C[p,s] \ \}$$
$$\text{gen\_set} = \{ \ C(p,s) \mid s \in H \land C[q,s] \ \} \cup \{ \ C(p,p) \mid C[q,q] \ \}$$

This rule is illustrated with the following example. Note that after executing the statement `p=q`, `p` is connected with everything that `q` was connected with before the statement.

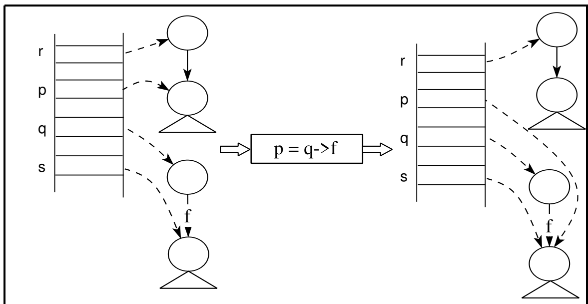


kill_set =
{ C(p,p), C(p,r) }

gen_set =
{ C(p,p), C(p,q), C(p,s) }

Note that if `q` presently points to `NULL`, `p` should also point to `NULL` after the statement. In this case all entries C[q,s] will be zero, resulting in an empty gen_set. Consequently all entries C_n[p,s] will also be zero after the statement, indicating `p` to be pointing to `NULL`, as desired. Similarly if `q` happens to be pointer `p` itself, resulting in the statement `p = p`, the gen and kill sets would be identical. In this case the connection matrix would remain unchanged, as required. Thus the above rule is general enough to take into account various special cases.

`p = q->f`: Pointer `p` now points to the heap object connected to the object pointed to by `q` through the pointer field `f`. Thus it points to the same data structure as `q`, even if not to the same heap object as `q`. So the analysis rule for this statement is same as that for the statement `p = q`. The effect of this statement on connection relationships is demonstrated as follows.



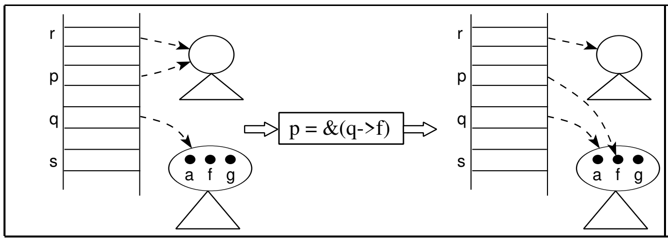
kill_set =
{ C(p,p), C(p,r) }

gen_set =
{ C(p,p), C(p,q), C(p,s) }

This rule incurs some imprecision, when the pointer `q->f` points to `NULL`. In this case, pointer `p` should also point to NULL after the statement. However we would report it be pointing to the same data structure as `q`. This information is *safe* but less precise. This happens because we cannot determine if `q->f` presently points to `NULL`, and not to a heap object. In other words, `q->f` is a heap-resident pointer, while connection matrix only abstracts the relationships (and nilness) of stack-resident pointers.

If pointers p and q are not distinct, the statement can be of the form `p = p->f`. The rule for this case is same as for the statment `p = p`, which does not change any connection relationships, as required.

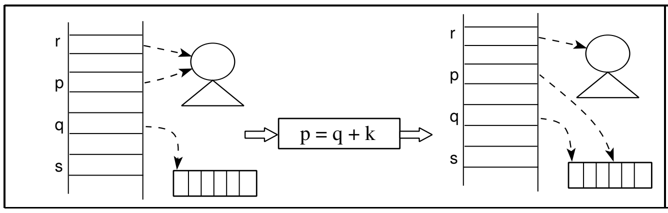
`p = &(q->f)` : Pointer p now points to the field f of the heap object pointed to by q, as shown in the following illustration. For the purpose of our analysis we consider a pointer pointing to a specific field of a heap object, to be pointing to the object itself. Thus, this statement is equivalent to the statement `p = q` for connection matrix analysis.



kill_set = { C(p,p), C(p,r) }

gen_set = { C(p,p), C(p,q) }

`p = q op k`: This rule represents pointer arithmetic. After the arithmetic operation, q continues to point to the same heap-object, though at a different offset, as shown in the following illustration. We assume that a heap-directed pointer does not cross the boundary of the heap object, when pointer arithmetic is performed on it. Otherwise, it can potentially point to memory not allocated by the program, and cause an execution error on being dereferenced. With this assumption about pointer arithmetic, this statement is equivalent to the statement `p = q` for connection matrix analysis.



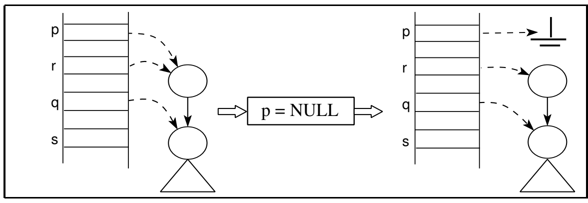
kill_set = { C(p,p), C(p,r) }

gen_set = { C(p,p), C(p,q) }

`p = NULL`: Pointer p now does not point to any heap object allocated by the program, as shown in the following illustration. It does not have any connection relationship with any pointer, including itself. Thus the effect of this statement is to simply kill all the relationships of p, as presented below:

$$\text{kill\_set} = \{\ C(p,s) \mid s \in H \wedge C[p,s]\ \}$$
$$\text{gen\_set} = \{\}$$

As illustrated in the following example, after executing the statement `p = NULL` we have C[p,p] = 0, indicating that p presently points to NULL.



kill_set =
{ C(p,p), C(p,r), C(p,q) }

gen_set =
{ }

2.3 Structure Updates

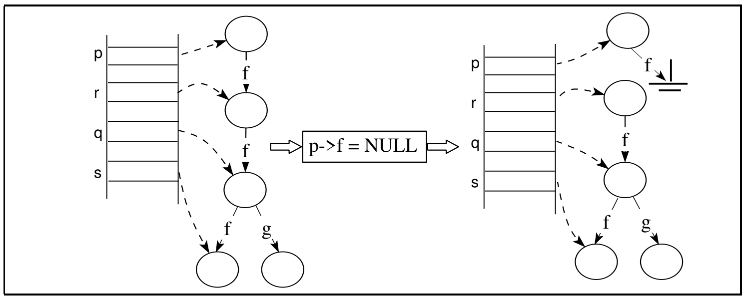
The statements discussed so far update a stack-resident heap-directed pointer. The following two statements update a pointer field residing in a heap object, and hence modify the structure of the heap itself.

`p->f = NULL`: This statement sets the field `f` to `NULL`. Consequently the subpiece pointed to by the pointer `p` gets disconnected from the remaining data structure. For example in the following illustration, after the statement `p->f = NULL`, pointer `p` does not have connection relationship with pointers `r`, `q` and `s`. However, to obtain this kill information we need to know the following:

- Does setting the field `f` to NULL, really disconnect a subpiece from the data structure? It is possible that the data structure still remains connected due to other links. For example in the following illustration, if pointers `p` and `r` are also connected through a `g` link, the subpiece pointed to by `r` would not get disconnected by the statement `p->f = NULL`.
- In case a subpiece gets disconnected, which pointers point to it?

Unfortunately, connection matrix information is not sufficient to answer these questions. To answer the first question we need to have some approximation for the shape of the underlying data structure. The second question requires knowledge about the possible path relationships between the various pointers pointing to the data structure. As such information is expensive to abstract, we do not collect it for level-1 heap analysis.

In the absence of precise kill information we err conservatively, and do not kill any connection relationships for this statement. Further, this statement does not generate any new relationships. Thus both the kill and gen sets are empty for this statement, and it does not affect connection relationships. This means that even though the real data structure is broken into two disjoint pieces (as illustrated in the picture), our connection matrix analysis cannot recognize this and will not be able to kill any connection relationships.



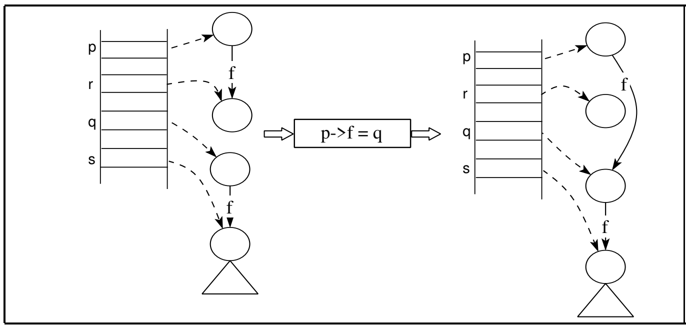
kill_set =
{ }

gen_set =
{ }

`p->f = q`: This statement has two effects. First it potentially disconnects a subpiece of the data structure pointed to by `p`, like the previous statement `p->f = NULL`. Next, it connects the data structures pointed to by `p` and `q`. As already discussed, precise kill information due to potential disconnection cannot be obtained. However new connection relationships are generated due to the interlinking of data structures pointed to by `p` and `q`. All pointers connected to `p` now get connected to all pointers connected to `q` (which include `q` itself). So we have the following analysis rule for this statement:

$$\text{kill\_set} = \{\}$$
$$\text{gen\_set} = \{\ C(r,s) \mid r,s \in H \wedge C[p,r] \wedge C[q,s]\ \}$$

This rule is illustrated in the following figure. In the real data structure, before the statement, pointers `p` and `r` are connected to `p`. After the assignment `p` is no longer connected to `r`, but it is now connected to `q` and `s`. However, note that the gen and kill sets for connection analysis cannot model this precisely. We cannot kill the connection between `p` and `r` and we also will generate a spurious connection between `r` and `s`. This happens because the disconnection of `r` from `p` cannot be inferred from the information available.



$\text{kill\_set} = \{\ \}$

$\text{gen\_set} = \{\ C(p,q),\ C(p,s),\ C(r,q),\ C(r,s)\ \}$

3 Implementing Connection Analysis in the McCAT C compiler

The connection matrix analysis has been implemented as a context-sensitive interprocedural analysis in the McCAT parallelizing/optimizing C compiler. The analysis is performed on the SIMPLE intermediate representation which is a simplified, compositional subset of C [Sri92, HDE+92, EH94]. The analysis is performed after points-to analysis and is implemented in a similar framework [Ema93, EGH94].

The implementation of connection matrix analysis is structured as a simple analysis for each basic statement of the form presented in Section 2, a compositional rule for each control construct, and an interprocedural strategy that uses an unfolded invocation graph to capture all calling contexts. Recursion is handled via special recursive and approximate nodes in the invocation graph, and an interprocedural fixed-point computation is performed at each recursive node.

Recursive nodes correspond to each point in the program where a recursive function is first called, and the approximate nodes correspond to all recursive calls that follow.

There are several important points in actually implementing this analysis. Firstly, one must be careful about how to apply the simple analysis rules presented in Section 2. The subtle point is that references of the form `p->f` may refer to the stack, the heap, or to both the stack and heap. For example, in one calling context, `p` may point to a stack-allocated object that has a name, while in another calling context `p` may point to a heap-allocated object. Consider a statement of the form `p->f = q`. If `p` points-to a stack-allocated object with the name `x`, then the appropriate connection analysis rule is `x.f = q`, whereas if `p` points-to a heap-allocated object, the appropriate rule is `p->f = q`. Thus, the actual implementation first uses the points-to information to resolve all references of the form `p->f` into a set of possible stack and heap locations, and then applies the appropriate simple connection analysis rules, merging the results of all the outputs.

The second important point is that a function `f` may change the connection relationships between variables that are not visible inside `f`. Thus, symbolic names must be generated to capture all such invisible locations. A method similar to the one used in the points-to analysis is used for this purpose.

The final important point relates to minimizing the overhead in getting an efficient context-sensitive analysis. We have currently implemented a simple memoization scheme that stores previously computed input and output values at each invocation graph node, and avoids recomputing the output values when a previously computed input/output pair can be reused. The usefulness of this scheme is discussed in Section 4.

4 Experimental Results

In this section, we present the experimental results obtained from connection matrix analysis of a set of 13 C programs. We chose programs that use a significant amount of dynamic allocation, as benchmarks for our study. In Table 1, we summarize the following characteristics for each program.

- Source lines including comments, counted using the *wc* utility.
- Number of statements in the SIMPLE intermediate representation. This number gives a good estimate of program size from the analysis point of view.
- Minimum, maximum and average number of variables abstracted by the connection matrices over all functions in the program (this includes symbolic variables introduced by our analysis). These numbers indicate the size of the abstraction and the memory requirements of the analysis for a given program.
- Total number of indirect references in the program, and the number of indirect references where the dereferenced pointer can point to a stack location, to a heap location and to both a stack and a heap location. The number of indirect references in a program, provides a measure for the relevance of pointer analysis to its optimization. The number of indirect references referring to stack and heap locations, respectively represent the significance of stack-based points-to analysis and heap-based data structure analyses for the given program.

The number of SIMPLE statements for the given benchmark set varies from 476 for *chomp* to 4909 for *volrend*, with an overall average of 2028 statements per program. The maximum number of variables abstracted by the connection matrix of a function is 133 for *pug*, followed by 114 for *cholesky*. The maximum of the average number of variables abstracted, is 89 for *cholesky* followed by 43 for *sim*. All the benchmarks have substantial number of indirect references, with maximum 822 for *pug* followed by 718 for *assembler*. Further, all of them have indirect references referring to both stack and heap locations, with the majority of indirect references referring to heap locations (except for the two benchmarks: *assembler* and *loader*). This makes the given benchmark set well-suited for evaluating a heap analysis.

As the analysis may become conservative when a pointer can point to both a stack and heap location, it is interesting to note that the right-most column indicates that this situation does not happen very often. We inspected the analysis output for programs *genetic*, *volrend* and *pug*, to detect the indirect references where this happens. We found that these indirect references mostly dereference formal parameters (of pointer type), to which both heap-directed and stack-directed pointers are passed as actuals, in different invocations of the given function.

Program	Source Lines	SIMPLE stmts	Min vars	Max vars	Avg vars	Ind Refs	To Stack	To Heap	Stack/ Heap
genetic	506	479	6	14	7	54	28	30	4
sim	1422	1760	38	69	43	374	34	340	0
blocks2	876	1070	28	54	33	373	98	275	0
ear	4953	3476	38	51	39	290	143	147	0
assembler	3361	3071	12	26	14	718	666	52	0
loader	1539	1055	7	20	10	170	106	64	0
cholesky	1899	2217	76	114	89	488	22	466	0
mp3d	1687	1849	18	28	20	490	25	465	0
water	2703	2418	8	65	27	581	32	549	0
volrend	4207	4909	18	45	20	190	63	128	1
chomp	430	476	20	27	22	127	45	82	0
sparse	2859	1495	12	40	18	468	3	465	0
pug	2400	2089	16	133	30	822	147	688	13

Table 1. Characteristics of Connection Matrix Benchmarks

4.1 Measurements for Heap Related Indirect References

In Table 2, we present empirical measurements for connection matrix analysis of the above benchmarks. Our measurements focus on indirect references in the program that refer to heap locations. We motivate our measurements using the following example program:

```
main()
  { p = my_malloc(N); q = my_malloc(M);
    ...
S:  *p = INIT_VAL;
T:  *q = INIT_VAL;
    ...  }
```

This program allocates two disjoint heap structures and then initializes them. Before connection analysis, the only information available from points-to analysis is: both the indirect references `*p` and `*q` (at statements S and T respectively) refer to the location *heap*, and thus the statements S and T interfere. After connection matrix analysis, we know that the data structures pointed to by `p` and `q` are never connected (are disjoint), and hence the statements S and T do not interfere.

Our experimental measurements attempt to quantify the improvement in resolution of heap data structures provided by connection matrix information over that obtained from the conservative approximation of points-to analysis. With only points-to analysis one must assume that each heap-directed pointer is possibly connected with all other other heap-directed pointers, while with connection matrix analysis one can identify a more precise set. Thus, the effectiveness of connection matrix analysis can be evaluated by comparing the total number of heap-directed pointers at an indirect reference (the conservative estimate provided by points-to analysis), with the total number of pointers connected with the dereferenced pointer (the more precise estimate available from connection matrix analysis). For example, in the above program, at statement S, the total number of heap-directed pointers is two (both `p` and `q` are heap-directed), while the number of pointers connected with the dereferenced pointer `p` is only one (`p` itself). The same situation holds at statement T.

Following this strategy, we have calculated the following metrics for each benchmark program (presented in Table 2):

refs: Total number of indirect references in the program that can refer to heap locations.

cavg: Average number of pointers that are connected with the dereferenced pointer at an indirect reference. This average is calculated as follows. At each indirect reference we determine the total number of pointers connected with the dereferenced pointer. Let us call this number `cn_tot_i` for the ith indirect reference in the program (as per lexical order). We do not include symbolic variables in this count as we generate them only to facilitate interprocedural mapping, and they cannot be accessed or dereferenced by the program. Further if the dereferenced pointer is only connected with itself, the count `cn_tot_i` will be one for the given indirect reference. We then sum up the numbers `cn_tot_i` for all indirect references, and divide this sum total denoted as `cn_sum_tot` by the total number of heap related indirect references in the program (**refs**), to obtain the average **cavg**.

havg: Average number of pointers that are heap-directed at an indirect reference. This average is calculated in the same fashion as **cavg**. First, at each indirect reference the total number of heap-directed pointers is calculated as `heap_tot_i`. Next, this number is summed up for all indirect references, and the sum total `heap_sum_tot` is divided by **refs** to obtain the average **havg**.

decr: A measure to approximate the percentage decrease in the number of connection relationships provided by connection matrix information over points-to information. It is calculated using the following formula: ((`heap_sum_tot` -

cn_sum_tot) * 100.0)/ (heap_sum_tot).

Without connection matrix analysis, the conservative approximation for the number cn_sum_tot would be simply heap_sum_tot, resulting in zero percentage decrease. With connection analysis, the more precise the analysis, the fewer the number of connection relationships reported, and the larger is the decrease. Thus, the metric **decr** provides a reasonable measure for the effectiveness of connection matrix analysis. For our small example program (given above): **refs** is 2, cn_sum_tot is 2 and hence **cavg** is 1.0; heap_sum_tot is 4, **havg** is 2.0 and **decr** is (((4 - 2) * 100.0) / 4) or 50%.

In Table 2, the left and middle parts present these measurements separately for indirect references of the type *a/(*a).b, and of the type a[i] where a is of pointer type. The right part of Table 2 gives the overall results. We discuss each case below.

Program	*a / (*a).b				a[i]				*a / (*a).b / a[i]			
	refs	cavg	havg	% decr	refs	cavg	havg	% decr	refs	cavg	havg	% decr
genetic	0	0.0	0.0	0.00	30	1.7	5.2	67.74	30	1.7	5.2	67.74
sim	96	3.4	23.2	85.55	244	1.6	20.4	92.41	340	2.1	21.2	90.29
blocks2	119	8.8	22.9	61.36	156	5.2	22.3	83.74	275	6.8	22.5	69.86
ear	42	2.7	3.8	27.22	105	2.4	7.1	66.26	147	2.5	6.1	59.40
assembler	45	4.4	7.8	42.98	7	6.0	9.4	36.36	52	4.6	8.0	41.93
loader	55	5.1	6.5	21.07	9	1.0	4.1	75.66	64	4.5	6.1	26.21
cholesky	82	14.9	34.3	56.46	384	3.7	20.7	82.27	466	5.7	23.1	75.53
mp3d	391	2.5	8.6	70.42	74	1.9	7.1	73.86	465	2.4	8.4	70.88
water	250	15.4	31.2	50.69	299	14.7	24.1	38.94	549	15.0	27.3	45.05
volrend	96	7.4	22.2	66.73	32	9.8	18.8	47.59	128	8.0	21.3	62.52
chomp	56	5.2	7.2	27.65	26	1.6	3.9	59.00	82	4.1	6.2	33.86
sparse	384	9.3	10.1	7.23	0	0.0	0.0	0.00	384	9.3	10.1	7.23
pug	514	36.8	36.9	0.30	174	47.6	47.7	0.11	688	39.5	39.6	0.24

Table 2. Empirical Measurements for Connection Matrix Results

Indirect References of type a[i]*:* The percentage decrease (**decr**) is, in general, higher for indirect references of this type. This happens because most of these references represent stack-based pointers that point to dynamically allocated memory and access it as an array (of non pointer type). For example, the statement `a = (int *) malloc(8 * sizeof(int))` dynamically allocates an array of eight integers. Such array structures are, in general, not pointed to by many other pointers. In SIMPLE, the above statement is *simplified* as `temp_0 = malloc(8 * sizeof(int)); a = (int *) temp_0`, resulting in both a and temp_0 pointing to the allocated structure. In case the allocation is done through a user-defined routine (for example `a = my_malloc(size)`) where type casting is not performed, the temporary variable is not generated, and pointer a alone points to the allocated structure. So the number of connection relationships of pointers like a tends to be close to 2.0 on an average. In Table 2, **cavg** for indirect array references is in the range of 1.0 to 3.7 for most of the benchmarks.

For some benchmarks **cavg** tends to be much larger. The benchmarks *volrend* and *blocks2* use arrays of pointers. Since we represent the entire array by the array name, connection relationships of pointers representing different indices of the array get merged. This results in large number of relationships for the single name representing them in the connection matrix. The benchmarks *assembler*, *water* and *pug* have pointers to arrays as fields of dynamically allocated structures (as opposed to being located on the stack). These pointers are reported to be connected with all other pointers that point to the given data structure. This results in larger overall **cavg** for these benchmarks. Actually **cavg** for *pug* is almost same as **havg**, as it builds only a single complex data structure, providing effectively no decrease in connections.

Indirect References of type `*a/(*a).b`*:* For indirect references of this form, the percentage decrease is, in general, not as high as for indirect array references. Such indirect references commonly access big aggregate data structures that consist of a large number of heap objects, specially if the data structure is recursive. Several pointers point to any such data structure, and all of them have connection relationships with each other.

In our benchmark set, *sim* and *mp3d* primarily use structures with no pointer fields. The percentage decrease for them is quite high, as these structures are also stand-alone entities in the heap, like dynamic arrays of non pointer type.

The benchmarks *ear* and *cholesky* primarily allocate structures with non-recursive pointer fields. For *ear*, **cavg** is quite small, though the percentage decrease is not very high as not many pointers are heap-directed in this program. For *cholesky* we have more than 50 percent decrease.

The benchmark *volrend* allocates integers and floats in the heap and accesses them through indirect references of the form `*a`. The percentage decrease for it could be even higher, but it uses arrays of pointers to point to heap-allocated integers and floats. The benchmark *blocks2* allocates several disjoint arrays of pointers to dynamically-allocated objects of type `int` and user-defined structure types with both recursive and non-recursive pointer fields. So it has higher **cavg**, but still shows substantial percentage decrease.

The benchmarks *assembler* and *loader* use two disjoint linked list data structures, *chomp* uses a linked list and a tree structure, while *water* uses arrays of linked lists several of which are disjoint at different points in the program. The percentage decrease statistics for these benchmarks show the following expected trend: the greater is the number of disjoint data structures used by a program, the better are the connection matrix results for it.

Finally, the programs *sparse* and *pug* use a single complex recursive data structure, and all heap-directed pointers point to it. Consequently, connection matrix analysis provides negligible improvement for them.

Overall results: In the third section of Table 2, we present the overall measurements for all the benchmark programs. The percentage decrease is highest for programs that primarily use dynamic arrays (of non pointer type) and structures without pointer fields (*sim*, *cholesky* and *mp3d*). For some programs (*genetic* and *ear*) the percentage decrease is not very high, but **cavg** is quite small which indicates that connection matrix analysis provides effective information for them. Overall, the results show that if the given program uses disjoint data structures, connection matrix analysis can always provide more accurate information for resolving heap related indirect references (as compared to the information provided by points-to analysis). Thus, the connection matrix abstraction works well for its target domain of applications, and more powerful and more costly analyses are required for other applications.

4.2 Interprocedural Measurements

Connection matrix analysis is a context-sensitive interprocedural analysis. In Table 3 we present some measurements demonstrating the interprocedural characteristics of the analysis. In Table 3(a), we provide some *static* interprocedural characteristics of the benchmarks. The first three columns in this table give the number of functions, call-sites, and nodes in the invocation graph. The last three columns give the number of recursive and approximate nodes, and the number of invocation graph nodes per call-site.

In Table 3(b), we provide some *dynamic* interprocedural measurements. The column labeled Tot gives the total number of procedure calls analyzed during the analysis. Our interprocedural algorithm analyzes a procedure once for each invocation context. Hence one might expect that the number of procedure calls analyzed would be equal to the number of nodes in the invocation graph. However, this is not true, since a procedure call can be analyzed several times for a single invocation context, if the call is involved in a loop or recursion fixed-point approximation. The column labeled Memo shows the number of procedure calls that get memoized using our simple memoization scheme, and the column labeled Actual gives the number of procedure calls that are not memoized. The last three columns, labeled Avgf, Avgc and Avgi, give the average number of calls actually analyzed (given in the column labeled Actual) per function, per call-site and per invocation graph node. These averages are calculated by dividing the number in the Actual column, with the appropriate number from the first three columns in Table 3(b). In other words, Avgf, Avgc and Avgi give the average number of times: (i) a function gets analyzed, (ii) a call-site is encountered during the analysis, and (iii) a call-chain in the program (possibly ending in recursion) is traversed during the analysis.

There are several interesting observations to be made from the results in Table 3. The first is that for these benchmarks, the size of the invocation graph does not explode, and we can do a complete context-sensitive analysis with reasonable cost. There are, however, other benchmarks that do have very large invocation graphs, so we are exploring more aggressive memoization techniques for handling these programs. It is also interesting to note that a large number of procedure calls get memoized, even with our simple scheme.

5 Conclusions

In this paper we have presented our approach to practical heap analysis for C. In contrast to other approaches that solve the stack-directed and heap-directed pointer problems simultaneously, we separate the two problems. The stack-based

Program	fns	call sites	ig nodes	Recur nodes	Appr nodes	nodes/ call	Calls Analyzed Tot	Memo	Actual	Avgf	Avgc	Avgi
genetic	17	32	45	0	0	1.41	55	8	47	2.76	1.46	1.04
sim	14	26	44	2	8	1.70	71	10	61	4.36	2.35	1.39
blocks2	20	28	28	1	2	1.00	371	221	150	7.50	5.36	5.36
ear	64	144	235	2	2	1.63	268	30	238	3.72	1.86	1.01
assembler	52	263	642	0	0	2.44	767	101	666	12.80	2.53	1.04
loader	30	82	125	2	2	1.52	312	132	180	6.00	2.20	1.44
cholesky	47	72	93	2	2	1.29	132	35	97	2.06	1.35	1.04
mp3d	23	28	32	0	0	1.14	47	0	47	2.04	1.68	1.47
water	15	21	26	0	0	1.24	98	73	25	1.67	1.19	0.96
volrend	53	108	169	2	2	1.56	192	21	171	11.40	1.58	1.01
chomp	22	47	98	7	7	2.09	219	92	127	5.77	2.70	1.30
sparse	28	76	121	0	0	1.59	168	47	121	4.32	1.60	1.00
pug	41	69	101	0	0	1.46	160	47	113	2.75	1.64	1.12

(a) Invocation graph characteristics (b) Interprocedural measurements

Table 3. Invocation graph characteristics and interprocedural measurements

problem is solved with points-to analysis. A hierarchical approach to the heap-directed pointer problem is used. For programs with few heap references, points-to analysis can be used directly (level-0 heap analysis). Points-to analysis gives a very conservative answer by treating the entire heap as one named location. For programs that allocate many disjoint structures, such as scientific programs with dynamically-allocated arrays, connection matrix analysis (level-1 heap analysis) provides useful information about the disjointness of the heap-allocated structures.

We have implemented the method in the McCAT compiler, and we provided experimental results to show that connection matrix analysis gives a substantially more precise answer than points-to analysis. The results are very good for the target application domain. For applications that use structures that are heavily linked, connection matrix analysis is not enough, and our approach has been extended to provide a more expensive direction/interference/shape analysis that allows us to estimate structure interference and estimate the shape (tree/dag/graph) of each structure allocated in the heap [Ghi95].

As the other implemented methods give experimental results for the stack and heap pointers combined, we do not have an empirical comparison. However, it is clear that connection analysis is much less sensitive to the location of `malloc` sites than any approach that uses these sites as names[CBC93, WL95]. In order to distinguish between different allocations, "malloc-site" based approaches must use call sites or call strings. This can easily result in a very large set of names in the abstraction, and thus slow down the analysis. In our case, we make use of the fact that each newly-allocated node is disjoint from all other heap-directed pointers live at that point in the program. Thus, we do not have to include many extraneous names in the abstractions.

Compared to Landi and Ryder's k-limiting approach [Lan92, LR92], we should get results similar to their analysis with k=1. The differences, if measured, would result from differences in where conservative approximations must be made in each method. Since the two approaches use different context-sensitive interprocedural approaches, there will be some variation. However, in our case we have the advantage of solving the heap problem separately, and thus the heap analysis operates on a relatively small set of pointers. Furthermore, our analysis fits cleanly into our hierarchical model where we have a family of heap analyses.

Our overall strategy is to use the analysis that is most appropriate for the target application program. If the program has no (or very little) dynamic allocation, then there is is no advantage to running the heap analysis, and points-to analysis is enough. For programs that allocate mostly non-recursive data structures, connection matrix analysis is simple and relatively inexpensive, while at the same time it provides useful results. We intend to continue working on the connection matrix analysis in order to further improve its efficiency, and to examine the benefit of context-sensitive vs. context-insensitive versions. We are also building tools for using connection matrix analysis to provide good interference analysis, particularly with respect to disambiguating arrays for array dependence analysis.

References

[CBC93] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In *Conf. Rec. of the Twentieth Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, Jan. 1993.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. of the SIGPLAN '90 Conf. on Programming Language Design and Implementation*, pages 296–310, White Plains, N. Y., Jun. 1990.

[Deu92] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *Proc. of the 1992 Intl. Conf. on Computer Languages*, pages 2–13, Oakland, Calif., Apr. 1992.

[Deu94] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pages 230–241, Orlando, Flor., Jun. 1994.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pages 242–256, Orlando, Flor., Jun. 1994.

[EH94] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proc. of the 1994 Intl. Conf. on Computer Languages*, pages 229–240, Toulouse, France, May 1994.

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill U., Montréal, Qué., Jul. 1993.

[Ghi95] Rakesh Ghiya. Practical techniques for interprocedural heap analysis. Master's thesis, School of Computer Science, McGill University, May 1995.

[Gua88] Vincent A. Guarna, Jr. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proc. of the 1988 Intl. Conf. on Parallel Processing*, volume II, pages 212–220, St. Charles, Ill., Aug. 1988.

[Har89] W. Ludwell Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation*, 2(3/4):179–396, 1989.

[HDE+92] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proc. of the 5th Intl. Work. on Languages and Compilers for Parallel Computing*, number 757 in Lec. Notes in Comp. Sci., pages 406–420, New Haven, Conn., Aug. 1992. Springer-Verlag. Publ. in 1993.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Parallel and Distrib. Systems*, 1(1):35–47, Jan. 1990.

[HPR89] Susan Horwitz, Phil Pfeiffer, and Thomas Reps. Dependence analysis for pointer variables. In *Proc. of the SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 28–40, Portland, Ore., Jun. 1989.

[JM81] N. D. Jones and S. S. Muchnick. *Program Flow Analysis, Theory and Applications*, chapter 4, Flow Analysis and Optimization of LISP-like Structures, pages 102–131. Prentice-Hall, 1981.

[JM82] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conf. Rec. of the Ninth Ann. ACM Symp. on Principles of Programming Languages*, pages 66–74, Albuquerque, N. Mex., Jan. 1982.

[Lan92] William A. Landi. *Interprocedural aliasing in the presence of pointers.* PhD thesis, Rutgers U., 1992.

[Lar91] J. R. Larus. Compiling Lisp programs for parallel execution. *Lisp and Symbolic Computation*, 4:29–99, 1991.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 21–34, Atlanta, Georgia, Jun. 1988.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. of the ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 235–248, San Francisco, Calif., Jun. 1992.

[PCK93] J. Plevyak, A. Chien, and V. Karamcheti. Analysis of dynamic structures for efficient parallel execution. In *Proc. of the 6th Intl. Work. on Languages and Compilers for Parallel Computing*, number 768 in Lec. Notes in Comp. Sci., pages 37–56, Portland, Ore., Aug. 1993. Springer-Verlag. Publ. in 1994.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, pages 13–22, La Jolla, Calif., Jun. 1995.

[Sri92] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill U., Montréal, Qué., Sep. 1992.

[WL95] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, pages 1–12, La Jolla, Calif., Jun. 1995.