Connection Analysis: A Practical Interprocedural Heap Analysis for C *

Rakesh Ghiya and Laurie J. Hendren
School of Computer Science, McGill University
Montréal, CANADA H3A 2A7
{ghiya,hendren}@cs.mcgill.ca

April 4, 1996

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 analyzing 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.

*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).

Keywords: Pointer Analysis, Heap Analysis, Pointer Disambiguation

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 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17], although more attention has been paid to actually implementing methods that work well for stack-allocated objects [10, 11, 15, 16, 17]. A complete discussion and comparison of these methods can be found in [18].

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`.

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* [10]. 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 [11] have been implementing a method for FORTRAN90, and they also compute aliases of pairs of *access paths*. Their access paths are similar to object names [10]. 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 [8]. 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 [16]. 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 [15, 19], 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 those functions which access heap-directed pointers (*heap* functions). We then evaluate the heap relationships only for this subset of heap-directed pointers, and only need to analyze the *heap* functions in the program.

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, and focuses on resolving stack

points-to relationships. 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. We further discuss related work in Section 5. Section 6 gives our conclusions and briefly describes our future work.

2 Connection Analysis

Connection analysis uses a simple, *storeless* [12], 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:

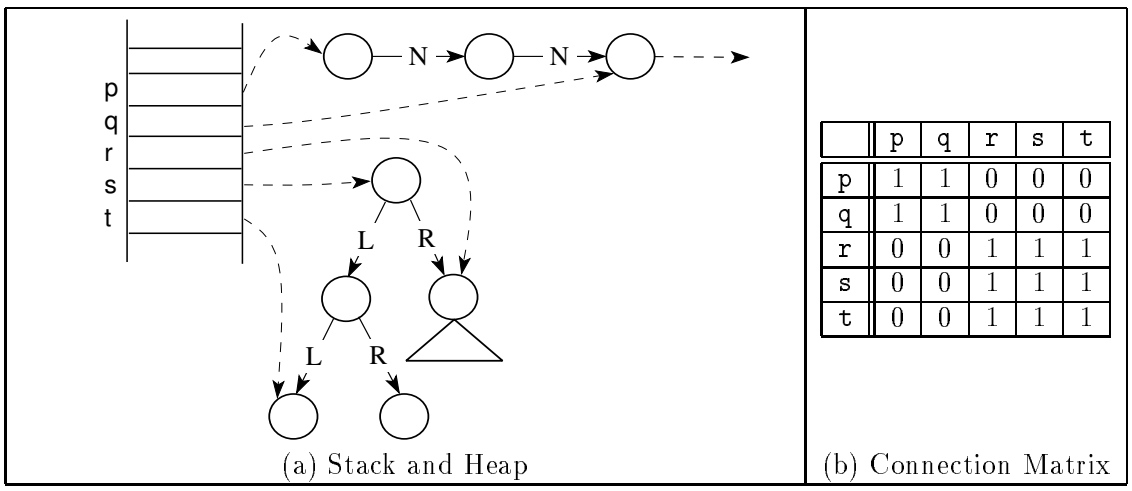
C[p,q] = 1: Pointers `p` and `q` *possibly* point to heap objects belonging to the same data structure. In our terminology, pointers `p` and `q` are considered to be *connected*, or to have a *connection* relationship.

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.



	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

Figure 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 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 statement, 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 statement (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.

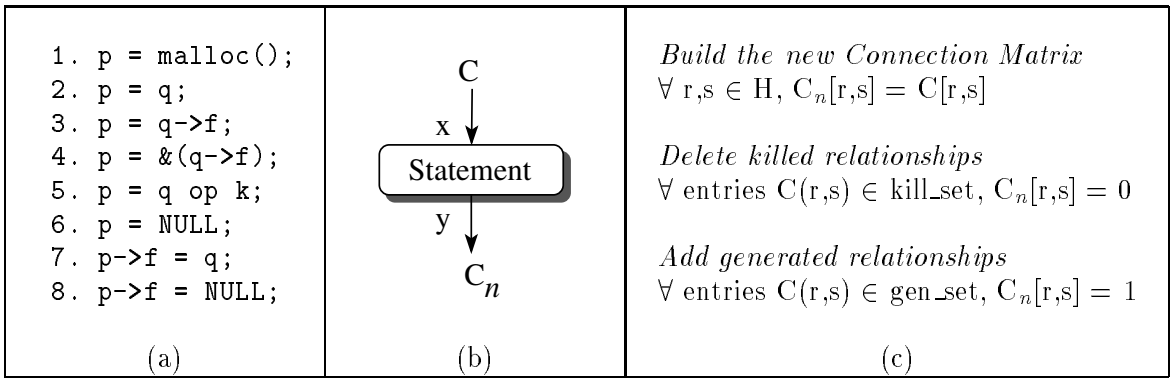


Figure 2: Computing Connection Matrix C_n from C for Basic Statements

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).

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.

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

$$\text{gen_set} = \{\ C(p,p)\ \}$$

The rule is illustrated in Figure 3. Note that after executing the `p = malloc()` statement, `p` is only connected with itself.

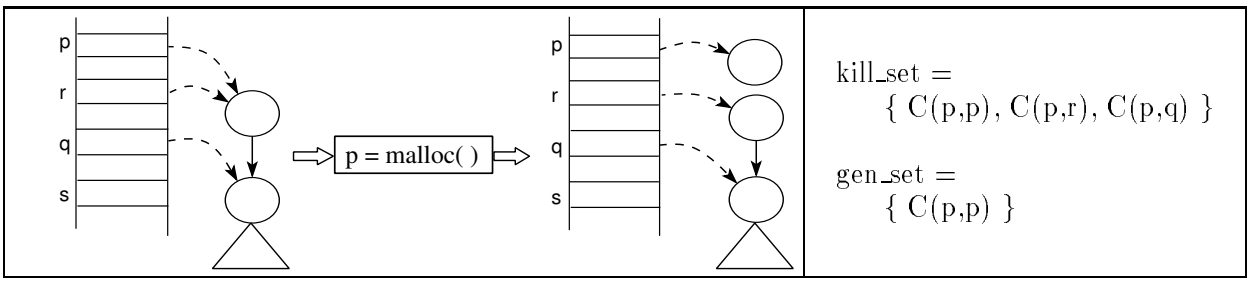


Figure 3: Example of analyzing `p = malloc();`

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.

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

$$\text{gen_set} = \{\ C(p,s) \mid s \in H \wedge C[q,s]\ \} \cup \{\ C(p,p) \mid C[q,q]\ \}$$

This rule is illustrated in Figure 4. Note that after executing the statement `p=q`, `p` is connected with everything that `q` was connected with before the statement.

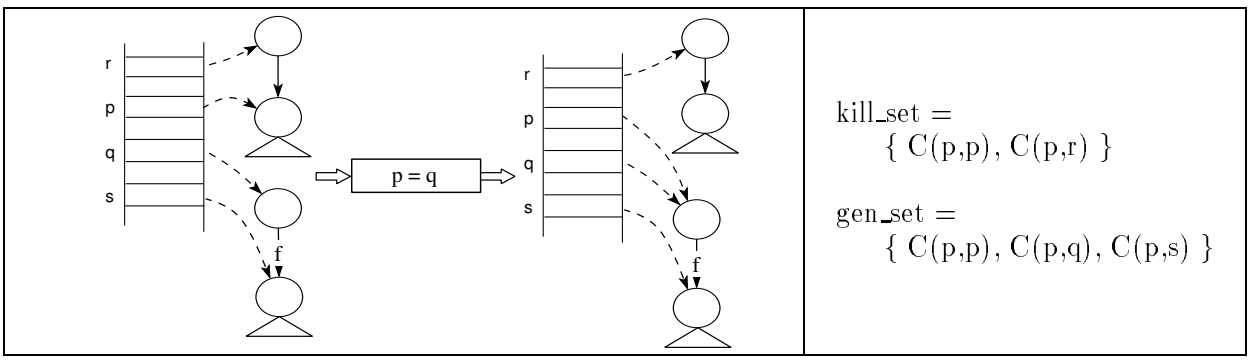


Figure 4: Example of analyzing `p = q;`

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 in Figure 5.

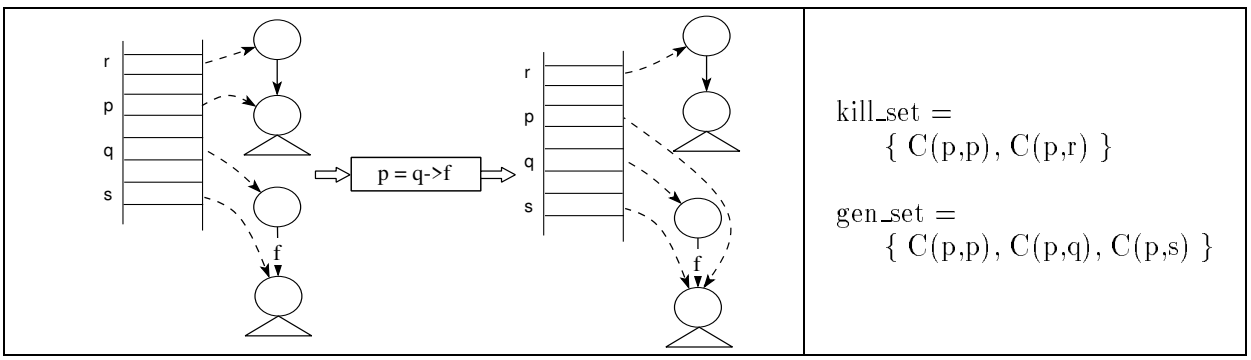


Figure 5: Example of analyzing `p = q->f`

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 statement `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 Figure 6. 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 analysis.

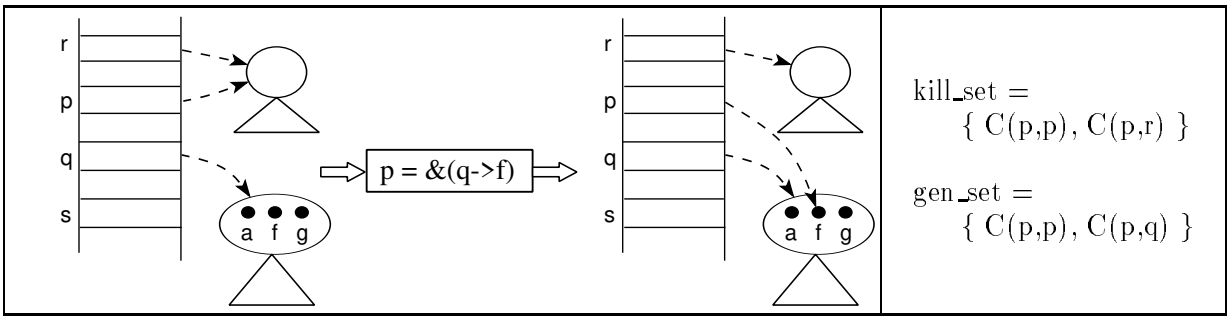


Figure 6: Example of analyzing `p = &(q->f)`

`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 Figure 7. 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 analysis.

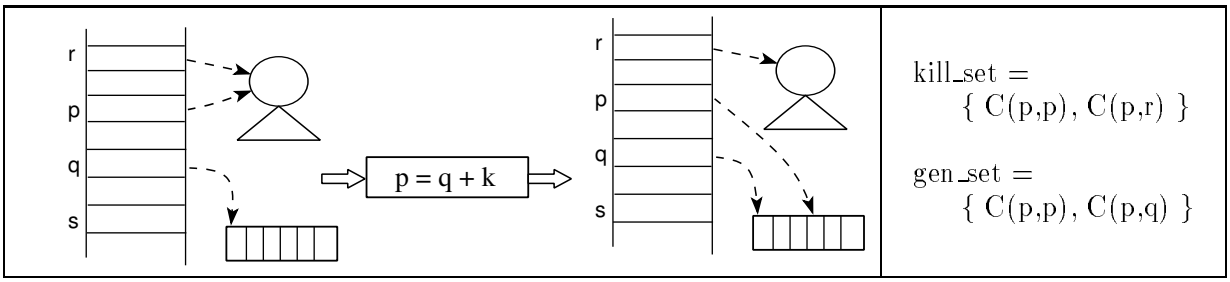


Figure 7: Example of analyzing `p = q + k`

`p = NULL`: Pointer `p` now does not point to any heap object allocated by the program, as shown in Figure 8. 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 Figure 8, after executing the statement `p = NULL` we have $C[p,p] = 0$, indicating that `p` presently points to NULL.

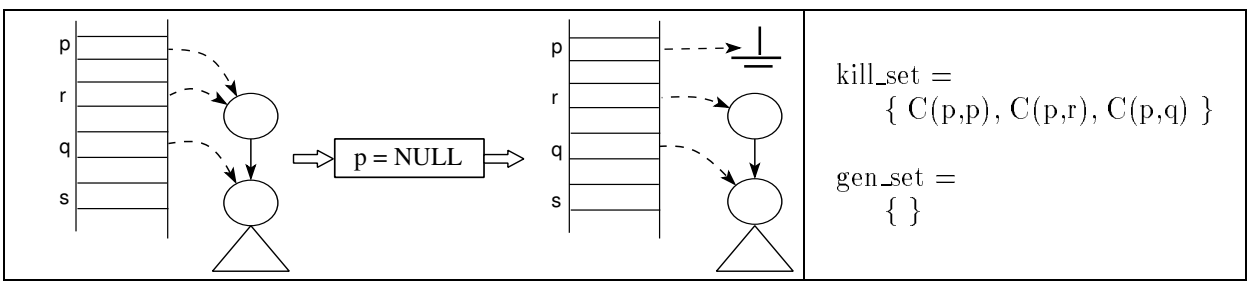


Figure 8: Example of analyzing `p = NULL`

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 Figure 9, 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 Figure 9, 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 Figure 9), our connection analysis cannot recognize this and will not be able to kill any connection relationships.

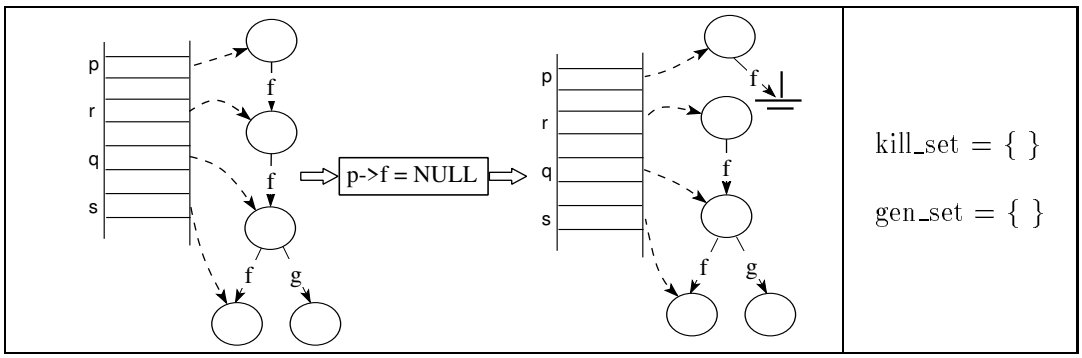


Figure 9: Example of analyzing `p->f = NULL`

`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 Figure 10. 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.

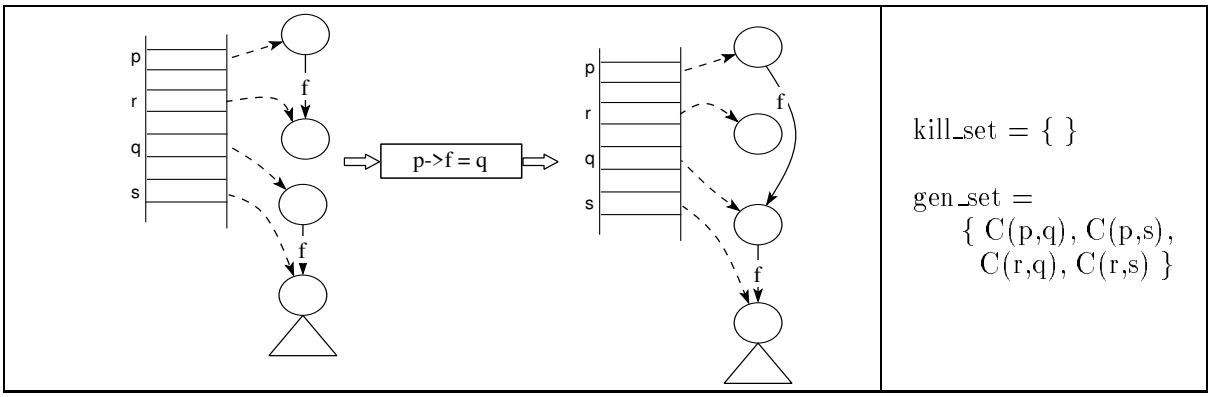


Figure 10: Example of analyzing `p->f = q`

2.4 Pointers from Heap To Stack

While defining the basic analysis rules, we had assumed that heap-resident pointers do not point to locations on the stack. With this assumption, if `p` points to a heap data structure, `p->f` should also point to a node in the same data structure. Without this assumption, `p->f` can also point to a stack location. The two cases are shown in Figure 11(with `N` denoting the field `f`). In part (b) of the figure, pointers `p` and `q` point to disjoint heap data structures from connection matrix point of view, as they are not linked by a heap-resident pointer. However, starting from pointer `p` one can access pointer `q`, and hence the data structure pointed to by `q`. On the contrary, we want that when `p` and `q` are not connected, `p` should not be able to access any heap location accessible from `q`, and vice versa.

Note that heap-resident pointers pointing to stack locations (henceforth we term these stack locations as *heap-pointed* locations), as such do not affect the correctness of connec-

tion analysis. Their presence just requires more careful interpretation of connection matrix information. Presently, we detect all heap-pointed locations by using points-to information: any pointer `x`, involved in points-to relationships of the form $(heap, x, P)$, falls into this category. In the presence of heap-pointed pointers, we ensure that any analysis or transformation using connection matrix information, makes the following conservative assumption: if a heap-pointed pointer is also heap-directed, the data structure pointed to by it can also be potentially accessed by any other heap-directed pointer.

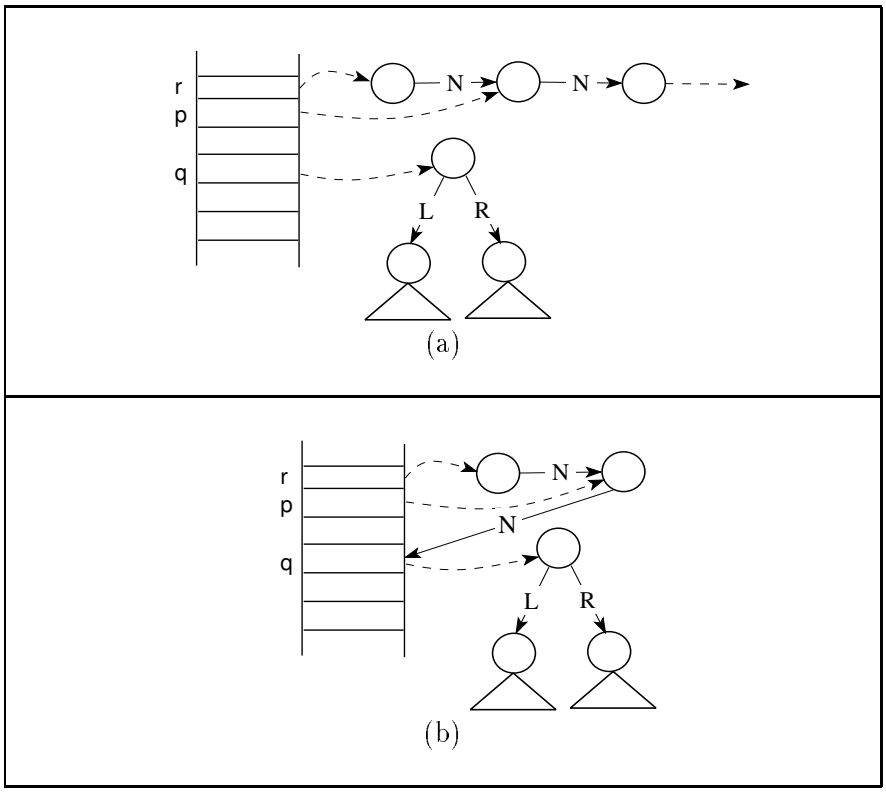


Figure 11: Handling Stack-connection Relationships

To enable more accurate assumptions, we plan to abstract another relationship called stack-connection. Pointer `p` is considered to be stack-connected with pointer `q`, if some heap

object belonging to the data structure pointed to by `p`, has a pointer field pointing to `q` i.e. the pointer field contains the address of `q`. Thus, in Figure 11(b), pointer `p` is stack-connected with pointer `q`. With this abstraction, we can state the following: Two heap-directed pointers cannot access a common heap location, if they are neither connected nor stack-connected.

To accurately capture stack-connection relationships, we need to build a stack-connection matrix for each function. This matrix abstracts relationships between heap-directed pointers and pointers which are reported to be heap-pointed by points-to analysis. A pointer becomes explicitly stack-connected with another pointer due to the statement `p->link = &q`. Further if a pointer `p` is stack-connected with another pointer `q`, all pointers connected with `p` also get stack-connected with `q`. Using these two basic rules, stack-connections can be computed in the same fashion as connection relationships, both intraprocedurally and interprocedurally.

In our experimental study of a collection of 12 C programs presented in Section 4, we found some programs to have heap-pointed stack locations. However, none of these locations turned out to be of pointer type. We plan to analyze a larger set of programs to evaluate, if it is worthwhile to separately abstract stack-connection relationships.

3 Implementing Connection Analysis in the McCAT C compiler

Connection 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 [20, 21, 22]. The analysis is performed after points-to analysis and is implemented in a similar framework [19, 15].

The implementation of connection 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 two optimizations to this end. First, we do a pre-pass over the program to identify functions that do not access any heap

location, and hence cannot affect connection relationships. We use points-to information to detect if any indirect access in a given function (and in any function called from it), can lead to the heap: if not, the function is flagged as a *non-heap* function. Connection analysis then analyzes the *sparser* program which contains only the *heap* functions. The second optimization consists of a powerful memoization scheme that stores pairs of previously computed input and output connection matrices for each *heap* function, and avoids re-analyzing the function body when the current input matrix matches a previously computed input matrix, by simply re-using the corresponding output matrix already stored. In Section 4 we provide concrete empirical data to demonstrate the effectiveness of these optimizations in improving the efficiency of the analysis without compromising its accuracy.

4 Experimental Results

In this section, we present the experimental results obtained from connection analysis of a set of 12 C programs. We chose programs that use a significant amount of dynamic allocation, as benchmarks for our study. Below we give a brief description of each benchmark program, and the principal data structures it uses:

- **genetic:** It implements a genetic algorithm to test sorting. The principal data structures used by this program are three global dynamically-allocated arrays of type `int`, which are also passed as parameters to various functions. Henceforth, we will refer to dynamically-allocated arrays as simply dynamic arrays.

- **sim:** This is a benchmark from computational biology that computes k-best non-intersecting alignments within a single DNA sequence or between two DNA sequences, using dynamic programming. The main data structures used by this program are

dynamic arrays of type `long`. It also uses dynamic arrays of pointers to structures. It allocates two types of structures: one with no pointer fields and one with a recursive pointer field.

- **blocks2:** This is another benchmark from computational biology that computes multiple aligned blocks from a given family of pairwise alignments for DNA sequences. It mainly uses dynamic arrays of type `long`. It also builds a constraint graph data structure using dynamic arrays of pointers to recursive structures.

- **ear:** This is a SPECINT92 benchmark that implements a model of acoustic propagation and detection in the human cochlea. It uses dynamic arrays and structures with non-recursive pointer fields.

- **assembler:** It implements an assembler and its principal data structures include: dynamic arrays and a linked list implementation for the symbol table.

- **loader:** It implements a loader, and uses the same data structures as the benchmark *assembler*. Both of these benchmarks are part of William Landi's test suite [10], and have been obtained from him.

- **cholesky:** It performs Cholesky factorization of a sparse positive definite matrix. It is part of the SPLASH [23] benchmark suite from Stanford. It implements the sparse matrix using structures with non-recursive pointer fields. These pointers point to dynamic arrays of type `int`.

- **mp3d:** This is another benchmark from the SPLASH suite related to rarefied fluid flow simulation used in aerospace research. It dynamically allocates structures with no pointer fields or with one non-recursive pointer field, and arrays of type `int` and

`float`.

- **water:** It solves the molecular dynamics N-body problem to evaluate forces and potentials in a system of water molecules in the liquid state, using spatial data structures. It is part of the new SPLASH benchmark suite called SPLASH-2 [24], and we use the sequential version. The primary data structures used by this program are linked lists and dynamically-allocated arrays of pointers pointing to linked lists.

- **volrend:** This benchmark renders a three-dimensional volume onto a two-dimensional plane using an optimized ray casting technique. It is also a part of the SPLASH-2 benchmark suite, and we analyze its sequential version. It dynamically allocates a number of bit vectors to store, manipulate and render the image. Its principal data structure is an array of pointers on the stack, which point to bit vectors allocated in the heap.

- **chomp:** It implements a game tree and uses two recursive data structures: a binary tree and a linked list, besides dynamic arrays.

- **sparse:** It builds a large and random sparse matrix using two-dimensional linked lists, then scales, factors and solves it. The sparse matrix data structure is a cyclic structure with nodes having links to nodes in the previous and next rows as well as columns.

In Table I, we further 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 478 for *chomp* to 4910 for *volrend*, with an overall average of 2025 statements per program. The maximum number of variables abstracted by the connection matrix of a function is 114 for *cholesky*, followed by 69 for *sim*. The maximum of the average number of variables abstracted, is 88 for *cholesky* followed by 43 for *sim*. All the benchmarks have substantial number of indirect references, with maximum 718 for *assembler* followed by 601 for *water*. 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 a 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*, *ear*, *mp3d* and *volrend*, to detect the indirect references where this happens. We found that

Program	Source Lines	SIMPLE stmts	Min vars	Max vars	Avg vars	Ind Refs	To Stack	To Heap	Stack/ Heap
genetic	506	481	6	14	7	54	27	30	3
sim	1422	1754	38	69	43	432	92	340	0
blocks2	876	1071	28	54	33	378	97	281	0
ear	4953	3485	38	51	39	292	147	147	2
assembler	3361	3071	12	26	14	718	666	52	0
loader	1539	1058	7	20	10	170	105	65	0
cholesky	1899	2218	76	114	88	508	22	486	0
mp3d	1687	1871	18	47	21	531	46	506	21
water	2703	2418	8	62	26	601	32	569	0
volrend	4207	4910	18	45	20	190	63	128	1
chomp	430	478	20	27	22	129	47	82	0
sparse	2859	1495	12	40	18	387	3	384	0

Table I: Benchmark Characteristics

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.

4.1 Measurements for Heap Related Indirect References

In Tables II and III, we present empirical measurements for connection 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 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 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 analysis one can identify a more precise set. Thus, the effectiveness of connection 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 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 Tables II and III):

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 *i*th 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 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 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 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 analysis, the conservative approximation for the number `cn_sum_tot` would be simply `heap_sum_tot`, resulting in zero percentage decrease. With connection anal-

ysis, 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 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 II, the left and right 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. Overall results are presented in Table III. We discuss each case below.

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 II, **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 con-

Program	`*a` / `(*a).b`				`a[i]`			
	refs	cavg	havg	% decr	refs	cavg	havg	% decr
genetic	0	0.0	0.0	0.00	30	1.7	5.2	67.74
sim	96	3.4	23.2	85.55	244	1.5	20.4	92.41
blocks2	125	8.6	22.7	61.83	156	5.3	22.4	76.43
ear	42	2.7	3.8	27.22	105	2.4	7.1	66.26
assembler	45	4.4	7.8	42.98	7	6.0	9.4	36.36
loader	56	5.2	6.5	20.82	9	1.0	4.1	75.68
cholesky	102	13.6	32.0	57.64	384	3.7	20.7	82.24
mp3d	432	3.9	10.4	62.91	74	10.1	16.0	36.49
water	270	15.8	31.1	49.11	299	14.8	24.1	38.68
volrend	96	7.4	22.2	66.73	32	9.8	18.8	47.59
chomp	56	5.2	7.2	27.65	26	1.6	3.8	59.00
sparse	384	9.3	10.0	7.23	0	0.0	0.0	0.00

Table II: Empirical Measurements for Connection Analysis Results

Program	`*a` / `(*a).b` / `a[i]`			
	refs	cavg	havg	% decr
genetic	30	1.7	5.2	67.74
sim	340	2.1	21.2	90.29
blocks2	281	6.8	22.5	69.89
ear	147	2.5	6.1	59.40
assembler	52	4.6	8.0	41.93
loader	65	4.6	6.2	25.87
cholesky	486	5.8	23.1	75.08
mp3d	506	4.8	11.2	57.41
water	569	15.3	27.4	44.29
volrend	128	8.0	21.3	62.52
chomp	82	4.1	6.2	33.86
sparse	384	9.3	10.0	7.23

Table III: Overall Empirical Measurements for Connection Analysis

nection matrix. The benchmarks *assembler* and *water* 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.

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 `long` 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 that connection analysis is also effective for benchmarks that use a number of disjoint recursive data structures.

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

Overall results: We now discuss the overall results presented in Table III. The percentage decrease is highest for programs that primarily use dynamic arrays (of non pointer type) and structures without pointer fields or with non-recursive 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 analysis provides effective information for them. Overall, the results show that if the given program uses disjoint data structures, connection 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

As reported in Section 3 connection analysis is a context-sensitive interprocedural analysis. In Tables IV and V we present some measurements for the interprocedural cost of the analysis, and the cost-reduction brought by the two interprocedural optimizations: (i) exclusion of *non-heap* functions, and (ii) memoization. In Table IV, statistics are presented both for the source and the sparse (excluding *non-heap* functions) versions of the program, by columns labeled respectively as 'src' and 'sps'. The left part of the table gives the number of functions

and the number of call-sites in the program. For the sparse version, only call-sites within *heap* functions are counted. Call-sites within *heap* functions which call *non-heap* functions are not counted. The right part of the table gives the number of procedure calls analyzed. Analyzing a procedure call involves analyzing the body of the called function in the given invocation context. The columns labeled 'Basic' and 'Memo' respectively give the number of procedure calls analyzed without and with the memoization optimization. The main observations from this table are as follows:

- A substantial number of functions (and hence call-sites) can be excluded from the analysis by identifying *non-heap* functions using the points-to information. For example, for benchmarks *ear*, *assembler* and *volrend* more than 50% functions can be excluded resulting in a much sparser program to analyze. The actual number of procedure calls analyzed also shows a significant reduction for the sparse program, for both the Basic and Memo analysis models. For *assembler* we see a reduction from 1642 to 406 for the Basic model and from 80 to 38 for the Memo model. These statistics indicate that separating stack (points-to) and heap analyses, enables a considerably more efficient heap analysis.

- The memoization optimization significantly reduces the interprocedural cost (due to context-sensitivity) of the analysis. For example, for the sparse version, the number of calls analyzed for *cholesky* goes down from 181 to 59, and for *water* from 145 to 12. This indicates that the actual number of different contexts with respect to an analysis, is much smaller than the number of static invocation contexts in the program.

In Table V, we provide more detailed interprocedural measurements for the case when both the *sparse* and the memoization optimizations are turned on. The first two columns in

Program	Funcs		Call sites		Calls Analyzed			
					Basic		Memo	
	src	sps	src	sps	src	sps	src	sps
genetic	17	11	33	19	70	31	22	14
sim	14	11	27	17	56	40	25	20
blocks2	20	16	29	22	288	274	51	46
ear	64	30	136	59	271	79	83	38
assembler	52	26	264	96	1642	406	80	38
loader	30	19	83	30	489	137	54	34
cholesky	48	45	73	65	189	181	62	59
mp3d	23	20	29	26	48	45	38	35
water	15	12	22	19	148	145	15	12
volrend	53	26	109	41	394	160	83	32
chomp	20	20	48	48	398	398	56	56
sparse	28	24	77	48	227	146	31	27

Table IV: Interprocedural Analysis Statistics

this table give the number of functions and the number of call-sites in the *sparse* version of the program (also given in Table IV). The third column labeled Total gives the number of procedure calls analyzed without memoization (same as the column 'sps' of the multicolumn 'Basic' in Table IV). The multicolumn labeled Memoized gives the number of procedure calls that benefit from memoization. These calls fall into two categories:

- Paid calls: When a procedure call is memoized, it involves the following check: if a precomputed input matrix matching the current input matrix exists for the called procedure. As there is some overhead involved in memoizing them, we term these calls as paid-memoized calls.

- Free calls: When a procedure call is paid-memoized, all the calls inside the called procedure get memoized for free, as the procedure body is not analyzed for the given context. These calls fall in the category of free-memoized calls.

The column Net gives the net number of calls for which the body of the called procedure is actually analyzed (same as the column 'Memo + sps' in Table IV). Finally, the columns Avgf and Avgc give the average number of calls actually analyzed (given in the column labeled Net) per function and per call-site. These averages are calculated by dividing the number in the Net column, with the appropriate number from the first two columns in the table.

In other words, Avgf gives the average number of times a function body gets analyzed, and Avgc gives the average number of times body of the called function is analyzed for a given call-site. An intraprocedural analysis analyzes each function body exactly once. So Avgf for it will be 1.0. Thus the deviation of Avgf from the value 1.0, indicates the extra cost incurred due to the interprocedural nature of an analysis. The Avgf for six of the benchmarks in Table V is less than 1.5, and only for two benchmarks it is greater than 2.0. The average

Program	funcs	call sites	Total	Memoized		Net	Avgf	Avgc
				Paid	Free			
genetic	11	19	31	9	8	14	1.27	0.74
sim	11	17	40	16	4	20	1.82	1.18
blocks2	16	22	274	149	79	46	2.88	2.09
ear	30	59	79	30	11	38	1.27	0.64
assembler	26	96	406	94	274	38	1.46	0.40
loader	19	30	137	57	46	34	1.79	1.13
cholesky	45	65	181	46	76	59	1.31	0.91
mp3d	20	26	45	4	6	35	1.75	1.35
water	12	19	145	56	77	12	1.00	0.63
volrend	26	41	160	26	102	32	1.23	0.78
chomp	20	48	398	98	244	56	2.80	1.17
sparse	24	48	146	54	65	27	1.12	0.56

Table V: Interprocedural Analysis Statistics with Memoization

value for Avgf over all the benchmarks is 1.64. These figures indicate that memoization brings the benefits of a context-sensitive interprocedural analysis with minimal overhead (for the given benchmark suite).

Another important observation from Table V is that Avgc for the majority of benchmarks is less than one. This happens because for some call-sites, the output matrix is already available from input/output matrix pairs stored (for memoization) during visits to the called function from previously processed call-sites. So the body of the called function is never analyzed with respect to these memoized call-sites.

Finally in Table VI we give the actual time (in seconds) taken by connection analysis when run on a *Sun Sparcstation 10*. It includes the time required for the pre-pass to identify *non-heap* functions (when applicable). The table gives timing data for all four varieties of connection analysis: without and with memoization (Basic and Memo), and without and with exclusion of *non-heap* functions (src and sps). The important observation from this table are:

- For the most efficient version of connection analysis (Memo + sps), it takes less than one second to analyze the majority of the benchmarks.

- Our study of the benchmarks suggests that analysis time is more dependent on program structure than program size. For example, the benchmarks *blocks2* (3.61 sec) and *cholesky* (3.46 sec) require more time than other larger benchmarks like *ear* and *water*, because they have calls to *heap* functions embedded in doubly nested loops (rendering the loop fix-point computation more expensive).

- For some benchmarks we see a significant decrease in analysis time when we shift from Basic to Memo model of the analysis, most notably for *assembler* and *volrend*. This happens because a large number of procedure calls get memoized for these

Program	Basic		Memo	
	src	sps	src	sps
genetic	0.10	0.06	0.06	0.05
sim	3.37	3.06	1.74	1.66
blocks2	5.82	5.61	3.62	3.61
ear	1.55	0.78	1.01	0.64
assembler	6.05	3.43	0.90	0.62
loader	1.17	0.53	0.46	0.36
cholesky	6.26	5.68	3.29	3.46
mp3d	0.39	0.37	0.36	0.34
water	3.51	3.41	1.72	1.69
volrend	33.75	30.96	1.02	0.89
chomp	1.54	1.49	0.50	0.45
sparse	1.48	1.33	0.69	0.60

Table VI: Analysis Time in seconds

benchmarks (Table V). Further for *volrend* calls to a recursive function (embedded insided a doubly nested loop) get memoized, which brings added benefit as a fix-point iteration is also required to estimate the output for a recursive function.

- For the *ear* benchmark, analysis with only *sparse* optimization (Basic + sps) takes less time than analysis with only memoization optimization (Memo + src). This happens because the *sparse* optimization for *ear* excludes 34 out of 64 functions, requiring only 79 calls to be analyzed as opposed to 83 for the only memoization option (Table IV). Thus both optimizations are important for reducing the interprocedural overhead of the analysis.

In this section, we have provided concrete data to demonstrate the effectiveness and efficiency of connection analysis, and to justify the strategy of separating stack and heap analyses. We are presently implementing dependence tests using connection information. Once this is achieved, we plan to measure the speed-up for the above benchmarks, due to the improved dependence information enabled by connection analysis.

5 Related Work

Our approach basically differs from other related work in three aspects: (i) decoupling of stack and heap analyses, (ii) design of a hierarchy of abstractions that suit different pointer analyses, and (iii) the use of stack points-to information to run a more efficient heap analysis (as evidenced by the data presented in Section 4).

Earlier work on pointer analysis focused solely on the analysis of heap-directed pointers (for languages like Lisp and Scheme). The basic approach was to approximate the structure of the heap in the form of a directed graph (with nodes as heap objects, and edges as links

between the objects). To keep the size of the graph finite, Jones and Muchnick [1] proposed the idea of *k-limiting* whereby all the nodes in the heap accessible from a variable after traversing k or more links, are coalesced into one summary node. Larus and Hilfinger [3] additionally labeled the nodes with access paths for dependence testing. Chase et. al [8] proposed the storage shape graph(SSG) which contains a node for each (heap-directed) pointer variable, and a node for each allocation site in the program. An allocation-site node in the SSG represents all the heap objects that can be allocated at that allocation site.

Hendren and Nicolau [9] departed from the graph-based approach, and proposed the path matrix abstraction, which captures the heap structure as path relationships between pointer variables (handles). Deutsch [12, 13] proposed a more powerful *storeless* model using pairs of *symbolic access paths* qualified by constraints that make them aliased. Recently, Sagiv et. al [25] have presented abstract storage graphs (ASGs) to accurately capture the heap structure in the presence of destructive updates.

The above approaches can provide a more refined estimation of heap structure as compared to connection analysis. For example, they can help identify 'treeness' or 'listness' of data structures. However they are also considerably more complex and difficult to implement in a real C compiler. On the contrary, connection analysis is the simplest analysis in a hierarchy of efficient heap analyses, specifically designed (and implemented) to disambiguate heap accesses at the data structure level. It also differs from the above-discussed methods, in being a heap analysis that also needs to take into account the presence of stack-directed pointers (using points-to information).

The recently proposed and implemented approaches for pointer analysis have mostly focused on the stack-directed pointer problem, using conservative approximations for the heap. Our points-to analysis uses only one abstract location *heap* to represent all heap objects. All

other methods use a variation of the allocation-site approach proposed by Chase et. al [8]. Landi and Ryder [10] and Ruf [17] simply name heap objects based on their allocation site (identified by calls to library routines like `malloc`). With this approach, if the programmer uses his own routine `my_malloc` with the `malloc` call embedded in it, all heap objects will get the same name. This would result in a similar scenario as our points-to analysis. To overcome this flaw, Choi et. al [11] proposed attaching procedure strings with the allocation site. Wilson and Lam [16] also follow the same approach. With this technique, heap objects allocated at the same allocation site but along different call-chains get distinct names. However this strategy sometimes results in a large set of names [16], which can slow down the analysis. Further this approach can still give the same name to completely unrelated heap objects.

Unlike the above methods, connection analysis is not sensitive to the location of allocation sites in the program. As explained in Section 2.1, it simply makes use of the fact that an allocation routine returns a new node not pointed to by any pointer in the program. Further if the call to an allocation routine is embedded inside a user-defined function, our analysis does not lose any precision because of its interprocedural nature. For example, suppose the user defines a function `my_malloc`:

```
void *my_malloc(int size)
{
  void *temp;
  temp = (void*) malloc(size);
  if (temp == 0)
    fatal_error("Virtual memory exhausted.");
  else
    return temp;
}
```

Now the statement `p = my_malloc(size)` will be analyzed as:

```
my_malloc(size);

p = return_my_malloc;
```

In the function call `my_malloc()`, statement `return temp` will be analyzed as:

```
return_my_malloc = temp;
return;
```

Thus after the function call, the global variable `return_my_malloc` will be pointing to the new heap object allocated by the call to `malloc()` in the function. The assignment `p = return_my_malloc` will make `p` also point to this object. Thus the statement `p = my_malloc(size)` has the same effect on connection relationships of `p` as the statement `p = malloc(size)`.

Another important point is that connection analysis reports two pointers to be connected, if they can point to the same heap data structure. While allocation-site based approaches can give the same name to completely unrelated heap objects: in our terms connect pointers which are actually 'disjoint'.

Landi and Ryder also combine allocation-site naming with k-limiting of object names (discussed in Section 1). This can sometimes provide better information than connection analysis. Consider the example below:

```
p = malloc();
p->link = malloc();
```

They would get the alias pairs `(*p, malloc_1)` and `(*(p->link), malloc_2)`, and can identify that pointers `p` and `p->link` lead to disjoint objects (of the same heap data structure). However this greatly depends on the structure of the given program, and no general conclusions can be drawn. On the other hand, connection analysis is our level-1 heap analysis, and is not designed to disambiguate heap accesses at this level. We have implemented a level-2 heap analysis called shape analysis [26] that can disambiguate accesses to the same heap data structure.

Finally as the other implemented methods give experimental results for the stack and heap pointers combined, we do not have a direct empirical comparison.

6 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 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 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 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 which are heavily linked, connection analysis is not effective, and our approach has been extended to provide a more expensive direction/interference/shape analysis that allows us to estimate the shape (tree/dag/graph) of each data structure allocated in the heap [18, 26] and disambiguate accesses to disjoint parts of the same data structure.

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, after points-to analysis. For programs that allocate mostly non-recursive data structures, connection analysis is simple and relatively inexpen-

sive, while at the same time it provides useful results. We are presently building tools for using connection analysis information to provide good dependence analysis, particularly with respect to disambiguating (dynamically-allocated) arrays for array dependence analysis.

References

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

[2] N. D. Jones and S. 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*, (Albuquerque, N. Mex.), pp. 66–74, Jan. 1982.

[3] J. R. Larus and P. N. Hilfinger, "Detecting conflicts between structure accesses," in *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, (Atlanta, Georgia), pp. 21–34, Jun. 1988.

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

[5] V. 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*, vol. II, (St. Charles, Ill.), pp. 212–220, Aug. 1988.

[6] S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence analysis for pointer variables," in *Proc. of the SIGPLAN '89 Conf. on Programming Language Design and Implementation*, (Portland, Ore.), pp. 28–40, Jun. 1989.

[7] W. L. Harrison III, "The interprocedural analysis and automatic parallelization of Scheme programs," *Lisp and Symbolic Computation*, vol. 2, no. 3/4, pp. 179–396, 1989.

[8] D. R. Chase, M. Wegman, and F. K. Zadeck, "Analysis of pointers and structures," in *Proc. of the SIGPLAN '90 Conf. on Programming Language Design and Implementation*, (White Plains, N. Y.), pp. 296–310, Jun. 1990.

[9] L. J. Hendren and A. Nicolau, "Parallelizing programs with recursive data structures," *IEEE Trans. on Parallel and Distrib. Systems*, vol. 1, pp. 35–47, Jan. 1990.

[10] W. Landi and B. G. Ryder, "A safe approximate algorithm for interprocedural pointer aliasing," in *Proc. of the ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, (San Francisco, Calif.), pp. 235–248, Jun. 1992.

[11] J.-D. 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*, (Charleston, South Carolina), pp. 232–245, Jan. 1993.

[12] A. 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*, (Oakland, Calif.), pp. 2–13, Apr. 1992.

[13] A. Deutsch, "Interprocedural may-alias analysis for pointers: Beyond k-limiting," in *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, (Orlando, Flor.), pp. 230–241, Jun. 1994.

[14] 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*, no. 768 in Lec. Notes in Comp. Sci., (Portland, Ore.), pp. 37–56, Springer-Verlag, Aug. 1993. Publ. in 1994.

[15] M. Emami, R. Ghiya, and L. 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*, (Orlando, Flor.), pp. 242–256, Jun. 1994.

[16] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for C programs," in *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, (La Jolla, Calif.), pp. 1–12, Jun. 1995.

[17] E. Ruf, "Context-insensitive alias analysis reconsidered," in *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, (La Jolla, Calif.), pp. 13–22, Jun. 1995.

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

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

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

[21] 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*, no. 757 in Lec. Notes in Comp. Sci., (New Haven, Conn.), pp. 406–420, Springer-Verlag, Aug. 1992. Publ. in 1993.

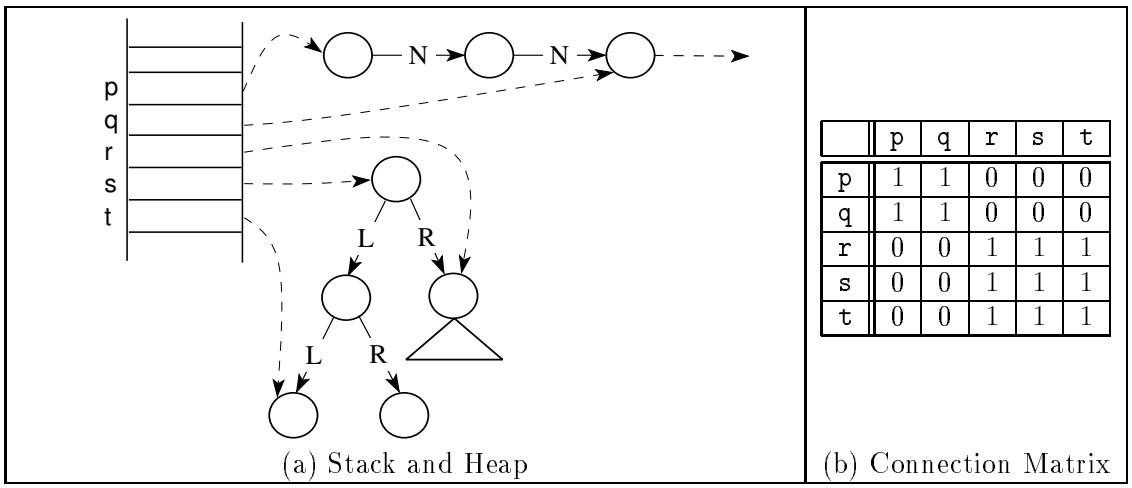
[22] A. M. Erosa and L. J. Hendren, "Taming control flow: A structured approach to eliminating goto statements," in *Proc. of the 1994 Intl. Conf. on Computer Languages*, (Toulouse, France), pp. 229–240, May 1994.

[23] J. P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," *Computer Arch. News*, vol. 20, pp. 5–44, Mar. 1992.

[24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. of the 22nd Ann. Intl. Symp. on Computer Architecture*, (Santa Margherita Ligure, Italy), pp. 24–36, Jun. 1995.

[25] M. Sagiv, T. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," in *Conf. Rec. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, (St. Petersburg, Flor.), pp. 16–31, Jan. 1996.

[26] R. Ghiya and L. J. Hendren, "Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C," in *Conf. Rec. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, (St. Petersburg, Flor.), pp. 1–15, Jan. 1996.



	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

Figure 1: An example Connection Matrix

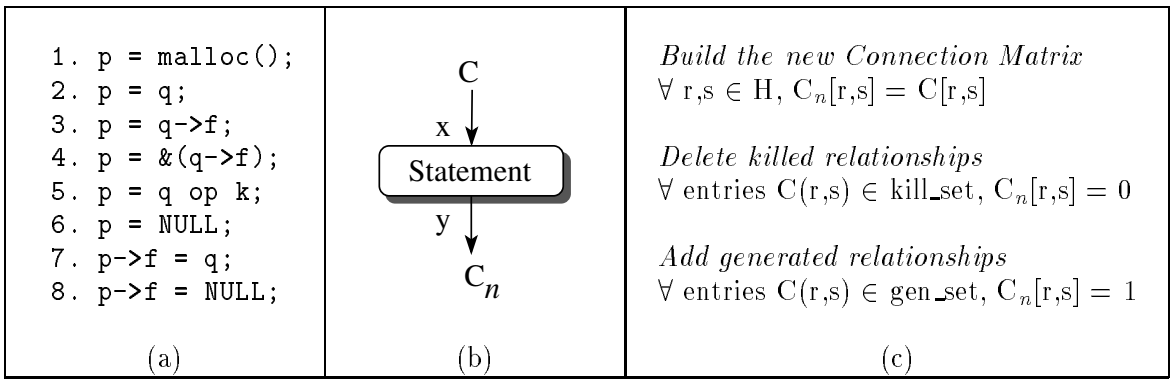


Figure 2: Computing Connection Matrix C_n from C for Basic Statements

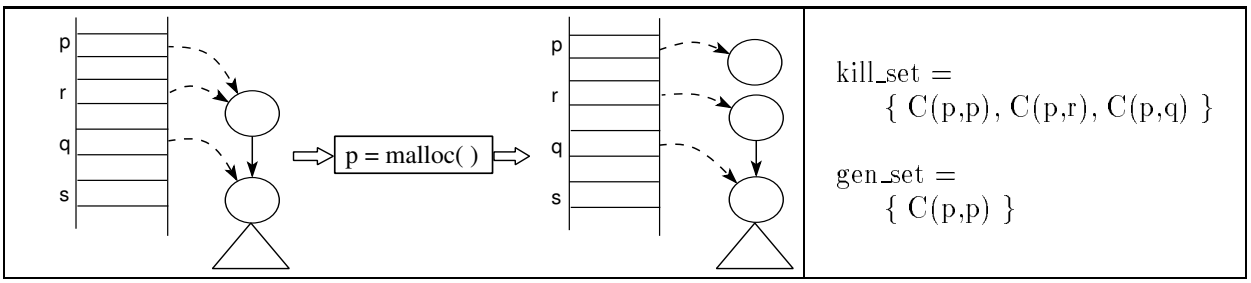
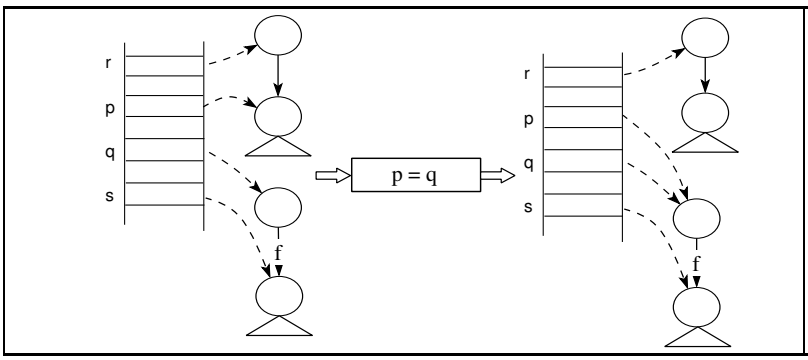


Figure 3: Example of analyzing `p = malloc();`



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

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

Figure 4: Example of analyzing `p = q;`

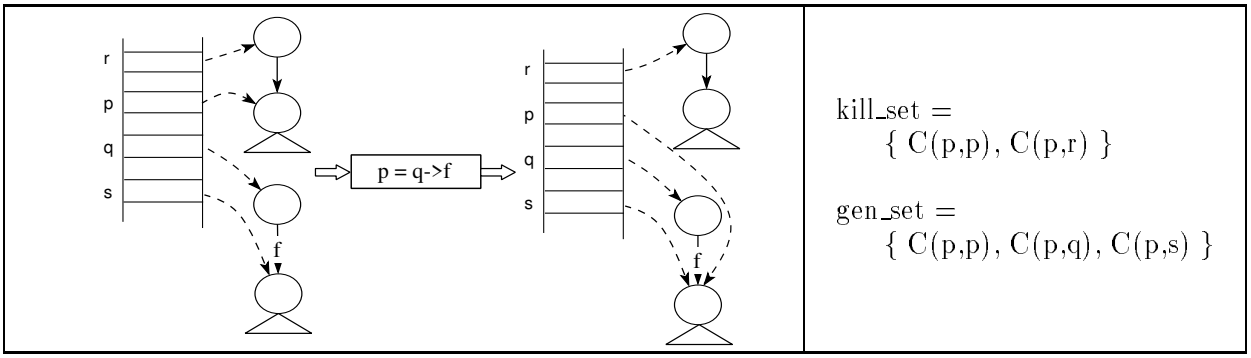


Figure 5: Example of analyzing `p = q->f`

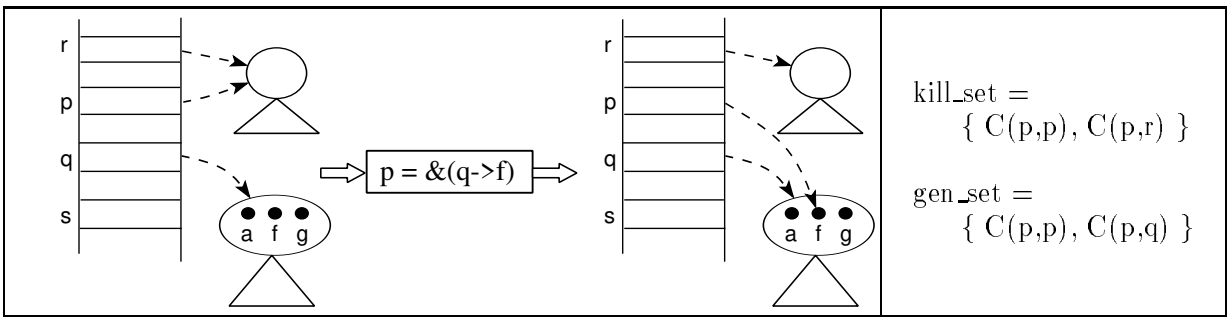


Figure 6: Example of analyzing `p = &(q->f)`

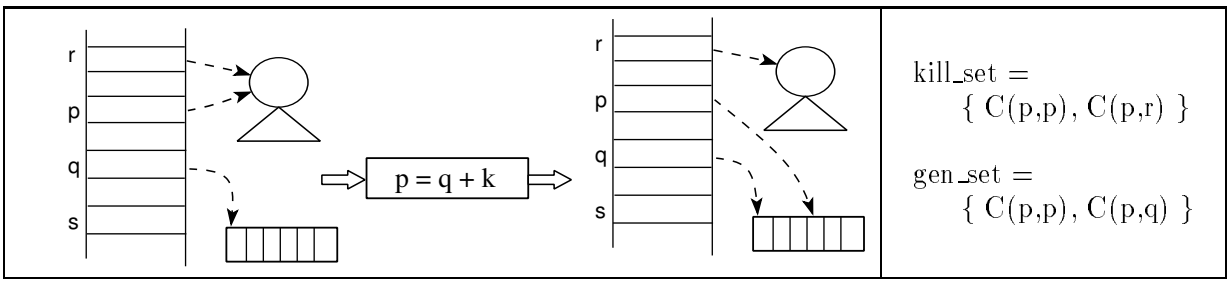


Figure 7: Example of analyzing `p = q + k`

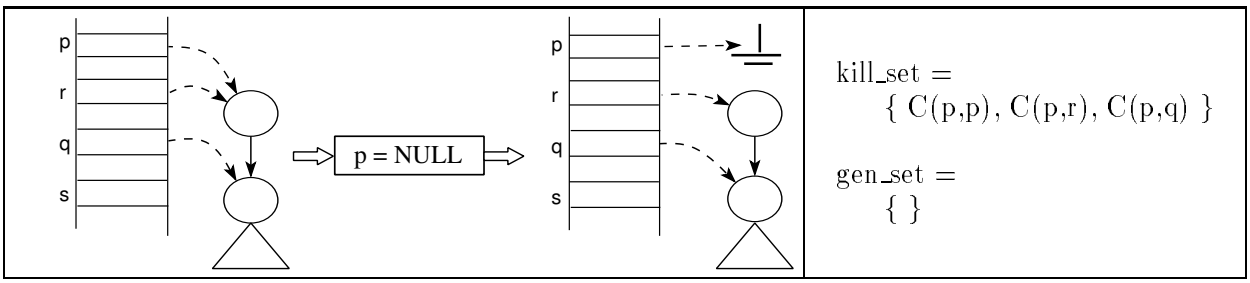


Figure 8: Example of analyzing `p = NULL`

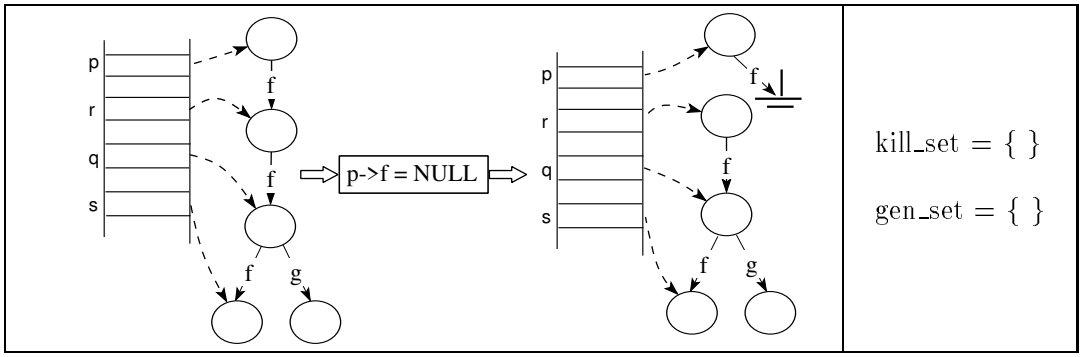


Figure 9: Example of analyzing `p->f = NULL`

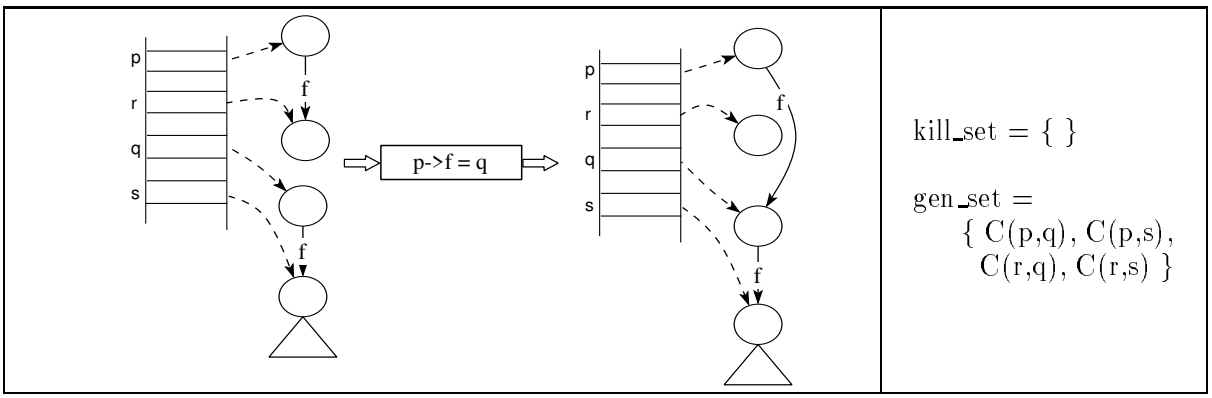


Figure 10: Example of analyzing `p->f = q`

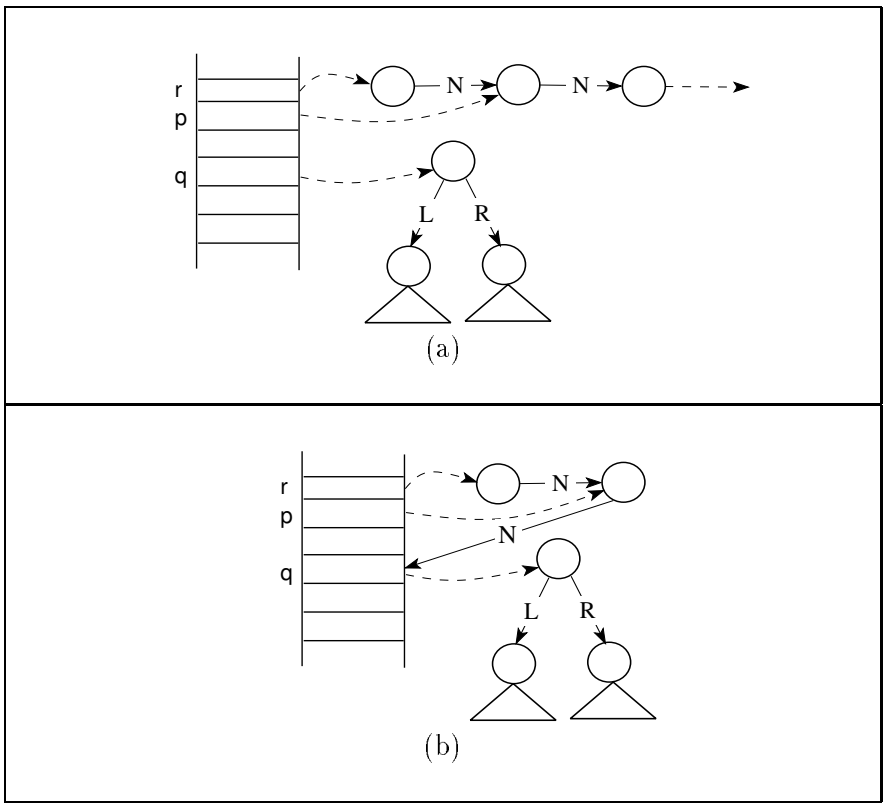


Figure 11: Handling Stack-connection Relationships

Program	Source Lines	SIMPLE stmts	Min vars	Max vars	Avg vars	Ind Refs	To Stack	To Heap	Stack/ Heap
genetic	506	481	6	14	7	54	27	30	3
sim	1422	1754	38	69	43	432	92	340	0
blocks2	876	1071	28	54	33	378	97	281	0
ear	4953	3485	38	51	39	292	147	147	2
assembler	3361	3071	12	26	14	718	666	52	0
loader	1539	1058	7	20	10	170	105	65	0
cholesky	1899	2218	76	114	88	508	22	486	0
mp3d	1687	1871	18	47	21	531	46	506	21
water	2703	2418	8	62	26	601	32	569	0
volrend	4207	4910	18	45	20	190	63	128	1
chomp	430	478	20	27	22	129	47	82	0
sparse	2859	1495	12	40	18	387	3	384	0

Table I: Benchmark Characteristics

Program	`*a` / `(*a).b`				`a[i]`			
	refs	cavg	havg	% decr	refs	cavg	havg	% decr
genetic	0	0.0	0.0	0.00	30	1.7	5.2	67.74
sim	96	3.4	23.2	85.55	244	1.5	20.4	92.41
blocks2	125	8.6	22.7	61.83	156	5.3	22.4	76.43
ear	42	2.7	3.8	27.22	105	2.4	7.1	66.26
assembler	45	4.4	7.8	42.98	7	6.0	9.4	36.36
loader	56	5.2	6.5	20.82	9	1.0	4.1	75.68
cholesky	102	13.6	32.0	57.64	384	3.7	20.7	82.24
mp3d	432	3.9	10.4	62.91	74	10.1	16.0	36.49
water	270	15.8	31.1	49.11	299	14.8	24.1	38.68
volrend	96	7.4	22.2	66.73	32	9.8	18.8	47.59
chomp	56	5.2	7.2	27.65	26	1.6	3.8	59.00
sparse	384	9.3	10.0	7.23	0	0.0	0.0	0.00

Table II: Empirical Measurements for Connection Analysis Results

Program	`*a` / `(*a).b` / `a[i]`			
	refs	cavg	havg	% decr
genetic	30	1.7	5.2	67.74
sim	340	2.1	21.2	90.29
blocks2	281	6.8	22.5	69.89
ear	147	2.5	6.1	59.40
assembler	52	4.6	8.0	41.93
loader	65	4.6	6.2	25.87
cholesky	486	5.8	23.1	75.08
mp3d	506	4.8	11.2	57.41
water	569	15.3	27.4	44.29
volrend	128	8.0	21.3	62.52
chomp	82	4.1	6.2	33.86
sparse	384	9.3	10.0	7.23

Table III: Overall Empirical Measurements for Connection Analysis

Program	Funcs		Call sites		Calls Analyzed			
					Basic		Memo	
	src	sps	src	sps	src	sps	src	sps
genetic	17	11	33	19	70	31	22	14
sim	14	11	27	17	56	40	25	20
blocks2	20	16	29	22	288	274	51	46
ear	64	30	136	59	271	79	83	38
assembler	52	26	264	96	1642	406	80	38
loader	30	19	83	30	489	137	54	34
cholesky	48	45	73	65	189	181	62	59
mp3d	23	20	29	26	48	45	38	35
water	15	12	22	19	148	145	15	12
volrend	53	26	109	41	394	160	83	32
chomp	20	20	48	48	398	398	56	56
sparse	28	24	77	48	227	146	31	27

Table IV: Interprocedural Analysis Statistics

Program	funcs	call sites	Total	Memoized		Net	Avgf	Avgc
				Paid	Free			
genetic	11	19	31	9	8	14	1.27	0.74
sim	11	17	40	16	4	20	1.82	1.18
blocks2	16	22	274	149	79	46	2.88	2.09
ear	30	59	79	30	11	38	1.27	0.64
assembler	26	96	406	94	274	38	1.46	0.40
loader	19	30	137	57	46	34	1.79	1.13
cholesky	45	65	181	46	76	59	1.31	0.91
mp3d	20	26	45	4	6	35	1.75	1.35
water	12	19	145	56	77	12	1.00	0.63
volrend	26	41	160	26	102	32	1.23	0.78
chomp	20	48	398	98	244	56	2.80	1.17
sparse	24	48	146	54	65	27	1.12	0.56

Table V: Interprocedural Analysis Statistics with Memoization

Program	Basic		Memo	
	src	sps	src	sps
genetic	0.10	0.06	0.06	0.05
sim	3.37	3.06	1.74	1.66
blocks2	5.82	5.61	3.62	3.61
ear	1.55	0.78	1.01	0.64
assembler	6.05	3.43	0.90	0.62
loader	1.17	0.53	0.46	0.36
cholesky	6.26	5.68	3.29	3.46
mp3d	0.39	0.37	0.36	0.34
water	3.51	3.41	1.72	1.69
volrend	33.75	30.96	1.02	0.89
chomp	1.54	1.49	0.50	0.45
sparse	1.48	1.33	0.69	0.60

Table VI: Analysis Time in seconds