

Efficient Context-Sensitive Pointer Analysis for C Programs

Robert P. Wilson and Monica S. Lam
Computer Systems Laboratory
Stanford University, CA 94305
<http://suif.stanford.edu>
{bwilson,lam}@cs.stanford.edu

Abstract

This paper proposes an efficient technique for context-sensitive pointer analysis that is applicable to real C programs. For efficiency, we summarize the effects of procedures using *partial transfer functions*. A partial transfer function (PTF) describes the behavior of a procedure assuming that certain alias relationships hold when it is called. We can reuse a PTF in many calling contexts as long as the aliases among the inputs to the procedure are the same. Our empirical results demonstrate that this technique is successful—a single PTF per procedure is usually sufficient to obtain completely context-sensitive results. Because many C programs use features such as type casts and pointer arithmetic to circumvent the high-level type system, our algorithm is based on a low-level representation of memory locations that safely handles all the features of C. We have implemented our algorithm in the SUIF compiler system and we show that it runs efficiently for a set of C benchmarks.

1 Introduction

Pointer analysis promises significant benefits for optimizing and parallelizing compilers, yet despite much recent progress it has not advanced beyond the research stage. Several problems remain to be solved before it can become a practical tool. First, the analysis must be efficient without sacrificing the accuracy of the results. Second, pointer analysis algorithms must handle real C programs. If an analysis only provides correct results for well-behaved input programs, it will not be widely used. We have developed a pointer analysis algorithm that addresses these issues.

The goal of our analysis is to identify the potential values of the pointers at each statement in a program. We represent that information using *points-to* functions. We consider heap-

This research was supported in part by ARPA contract DABT63-94-C-0054, an NSF Young Investigator award, and an Intel Foundation graduate fellowship.

In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 18–21, 1995, pp. 1–12. Copyright © 1995 by ACM, Inc.

allocated data structures as well as global and stack variables, but we do not attempt to analyze the relationships between individual elements of recursive data structures.

Interprocedural analysis is crucial for accurately identifying pointer values. Only very conservative estimates are possible by analyzing each procedure in isolation. One straightforward approach is to combine all the procedures into a single control flow graph, adding edges for calls and returns. An iterative data-flow analysis using such a graph is relatively simple but suffers from the problem of *unrealizable paths*. That is, values can propagate from one call site, through the callee procedure, and back to a different call site. Some algorithms attempt to avoid unrealizable paths by tagging the pointer information with abstractions of the calling contexts [2, 12]. However, these algorithms still inappropriately combine some information from different contexts.

Emami et al. have proposed a *context-sensitive* algorithm that completely reanalyzes a procedure for each of its calling contexts [6]. This not only prevents values from propagating along unrealizable paths, but also guarantees that the analysis of a procedure in one calling context is completely independent of all the other contexts. A procedure may behave quite differently in each context due to aliases among its inputs, and a context-sensitive analysis keeps those behaviors separate. Reanalyzing for every calling context is only practical for small programs. For larger programs, the exponential cost quickly becomes prohibitive.

Interval analysis, which has been successfully used to analyze side effects for scalar and array variables in Fortran programs [7, 10], is an approach that combines context sensitivity and efficiency. This technique summarizes the effects of a procedure by a *transfer function*. For each call site where the procedure is invoked, it computes the effects of the procedure by applying the transfer function to the specific input parameters at the call site. This provides context sensitivity without reanalyzing at every call site.

Interval analysis relies on being able to concisely summarize the effects of the procedures. Unfortunately, pointer analysis is not amenable to succinct summarization. The effects of a procedure may depend heavily on the aliases that hold when it is called. Thus, the evaluation of a transfer function that summarizes the pointer assignments in a procedure

may be no simpler than running an iterative algorithm over the original procedure.

We propose a new technique that is completely context-sensitive yet does not sacrifice efficiency. Our approach is based on the insight that procedures are typically called with the same aliases among their inputs. Thus it is not necessary to completely summarize a procedure for all the potential aliases but only for those that occur in the program. Our idea is to generate *incomplete* transfer functions that only cover the input conditions that exist in the program. These incomplete transfer functions are made up of simple *partial* transfer functions (PTFs) that are only applicable to calling contexts that exhibit certain alias relationships. We have developed an efficient technique that isolates the set of relevant aliases upon which a PTF definition is based.

Our analysis embraces all the inelegant features of the C language that are hard to analyze but are commonly used. It safely handles arbitrary type casts, unions, and pointer arithmetic. It also assumes that pointers may be stored in any memory locations, regardless of the types declared for those locations. Since some of the standard library functions may change the values of pointers, we provide the analysis with a summary of the potential pointer assignments in each library function.

We have implemented our algorithm in the SUIF compiler system and have measured the analysis times for a set of benchmark programs. Our empirical results show that our technique is successful in that it often needs to generate only one PTF for each procedure in the program.

This paper is organized as follows. We first introduce the major concepts behind our approach and give an outline of our algorithm in Section 2.1. We then describe our representation of memory locations and pointer values in Section 3. Next in Section 4, we explain the intraprocedural portion of our algorithm. Section 5 then describes how we compute the effects of procedure calls. Finally, we discuss related work in Section 6 and present our experimental results in Section 7.

2 Major Concepts

This section describes the major concepts in the design of our pointer analysis algorithm. We first introduce the general approach of using partial transfer functions as an efficient means to provide context sensitivity. We next describe the specific design of partial transfer functions that we use in our algorithm. Finally, we provide a complete outline of our algorithm.

2.1 Partial Transfer Functions

To provide some insight on how one might define transfer functions for pointer analysis, consider an informal summary for the very simple procedure `f` in Figure 1:

The target of `p` points to whatever the target of `q` initially pointed to.

```

f (p, q, r) {
    *p = *q;
    *q = *r;
}

int x, y, z;
int *x0, *y0, *z0;

main () {
    x0 = &x; y0 = &y; z0 = &z;
    if (test1)
S1:   f(&x0, &y0, &z0);
    else if (test2)
S2:   f(&z0, &x0, &y0);
    else
S3:   f(&x0, &y0, &x0);
}

```

Figure 1: Example Program

Case I If `r` and `p` do not point to any of the same locations, the target of `q` points to whatever the target of `r` initially pointed to.

Case II If `r` and `p` definitely point to exactly the same location, then the target of `q` retains its original value.

Case III If `r` and `p` may point to the same locations but their targets are not definitely the same, then the target of `q` may either retain its original value or point to whatever the target of `r` pointed to initially.

This example illustrates two important points. First, the aliases among the inputs to a procedure determine its behavior. Given a particular set of aliases, summarizing a procedure is relatively easy. Second, even for this simple two-statement procedure the complete summary is fairly complex. Computing a full summary for a procedure with cycles and recursive data structures is prohibitively expensive.

The main idea behind our algorithm is that we need not summarize a procedure for all possible aliases among its inputs. Instead, we only need to find summaries that apply to the specific alias patterns that occur in the program. For our simple example, Case I applies to the calls at S1 and S2 because they have the same alias pattern, even though they have totally different parameters. Case II applies to the call at S3. Thus it is not necessary to consider Case III for this particular program.

Our basic approach and its comparison with traditional interval analysis are illustrated in Figure 2. The traditional approach is to define a complete transfer function that maps the entire input domain for a procedure to the corresponding outputs (Figure 2(a)). Instead, we develop a set of partial transfer functions (PTFs), each of which is applicable to only a subset of the input domain. As shown in Figure 2(b), the domains of these partial functions are disjoint and the union of their domains does not necessarily cover all the possible

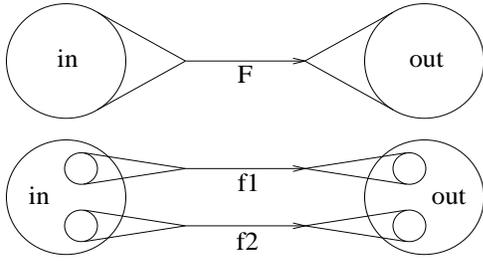


Figure 2: (a) Transfer function for a procedure and (b) partial transfer functions

inputs. Many potential inputs never occur in practice, and we only need PTFs for the inputs that actually occur. That means the complexity of all the PTFs taken together is much lower than that of the full transfer function.

2.2 Design of PTFs

There is a trade-off between the complexity of the individual PTFs and their applicability. By making a PTF more complicated, we can increase the size of its input domain so that fewer PTFs need to be computed. The complete transfer function, which covers all the possible inputs, is at one end of this trade-off. At the other extreme, each point in the input space can have a separate PTF. One of these PTFs can only be reused when all of its inputs have exactly the same values as in the context where it was created. The initial values specify the input domain for the PTF. Whenever the analysis encounters another call to the procedure with the same input values, it can reuse the final values recorded in the PTF. This technique is commonly known as *memoization*.

Since it is not the specific input values but their alias patterns that determine the behavior of a procedure, we use symbolic names called *extended parameters* to abstract the input values. An extended parameter represents the locations reached through an input pointer at the beginning of a procedure. Every object is represented by at most one extended parameter. This is similar to the “invisible variables” defined by Emami et al. [6].

For procedure f in Figure 1, we use the extended parameter 1_p to represent the location initially pointed to by p ; 1_p represents x_0 for the call at S_1 and z_0 for the call at S_2 . For the calls at S_1 and S_2 in Figure 1, since none of the inputs are aliased, we create a new extended parameter for every location accessed. For the call at S_3 on the other hand, both p and r point to the same location, so we create only one extended parameter to represent the common location.

When a pointer to a global is passed into a procedure, we treat it as any other extended parameter. We do not necessarily want to know the specific value of the pointer. For example, for the call at S_1 in Figure 1, the parameter 1_p represents the global variable x_0 . If we used x_0 directly instead of the parameter, we would not be able to reuse the same PTF for the call at S_2 . Since extended parameters

represent global variables referenced through pointers, we also use extended parameters to represent global variables that are referenced directly. If a global is referenced both directly and through a pointer input to a procedure, using the same extended parameter for both references takes care of the alias between them.

It is important to only create the extended parameters that are relevant to the procedure. Aliases involving parameters that are never referenced should not prevent us from reusing PTFs. Our solution is to create the extended parameters lazily. We only create extended parameters as they are referenced. In contrast, Emami et al. create invisible variables for all input pointers that could potentially be accessed. Not only does that require unnecessary overhead to create parameters that are never referenced, but it also limits the opportunities for reuse.

Extended parameters play three important roles:

1. **Extended parameters make up part of the name space of a procedure.** Each procedure has its own distinct name space which consists of the extended parameters, local variables, and heap storage allocated by the procedure and its children. We derive a *parameter mapping* at each call site to map the caller’s name space to the callee’s and vice versa.
2. **The initial points-to function for a PTF specifies the input domain.** The aliases among the inputs to a PTF are the main part of the input domain specification. With our definition of extended parameters, the initial points-to function succinctly captures that information. For the alias in the call at S_3 in the example, the initial points-to function records that p and r point to the same extended parameter 1_p as shown in Figure 4(a). Similarly, the totally disjoint points-to relationships in Figure 3(a) reflect the lack of aliases for the other calls.

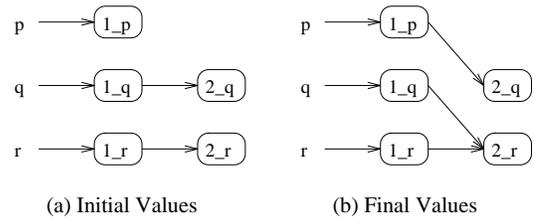


Figure 3: Parameterized PTF for Calls at S_1 and S_2

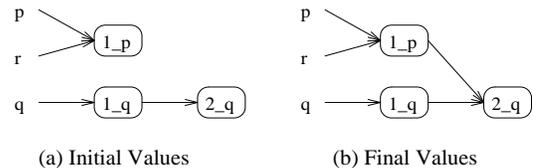


Figure 4: Parameterized PTF for Call at S_3

3. **The final points-to function at the procedure exit summarizes the pointer assignments.** Given the initial points-to function, it is relatively easy to derive the points-to function at the procedure exit. Figures 3(b) and 4(b) show the final points-to functions produced for the corresponding inputs. Since the analysis operates on the parameterized name space, the final points-to function summarizes the procedure parametrically. The parameter mapping for a particular call site allows us to translate the summary back to the caller’s name space.

Besides the aliases, the input domain of a PTF is also defined by the values of function pointers passed in and used in calls within the procedure. The function pointer values affect the PTF summary because they determine what code could be executed.

The PTF design presented here is but one choice in the trade-off between the complexity of the PTFs and the sizes of their input domains. We have also explored another scheme that uses a separate extended parameter for each access path. That design increases reuse with considerable cost in complexity, since multiple extended parameters may then refer to the same location. Furthermore, it requires additional analysis to abstract the potentially infinite access paths to a finite set. Our experience with that scheme suggests that such complexity is unnecessary for this analysis.

2.3 Algorithm Outline

Just as we create extended parameters lazily, we only create PTFs as they are needed. In this way, we do not compute unnecessary summaries. We begin by using an iterative data-flow approach to find the potential pointer values in the `main` procedure. When this iterative analysis encounters a call to a procedure with new input aliases, it recursively analyzes that procedure for the current context to produce a new PTF. We use a stack to keep track of the current calling contexts.

We update the initial points-to functions and parameter mappings lazily during the iterative analysis. When we begin analyzing a procedure, the initial points-to function is empty and the parameter mapping only records the actual values for the formal parameters. When we need to know the initial value of an input pointer and it is not already recorded, we add an entry to the initial points-to function. To check for aliases, we need to look up the initial values in the calling context. If the pointer has not yet been referenced in the calling context either, we will add an entry to the caller’s initial points-to function. The process continues recursively up the call graph until the pointer values are known. If the initial values are not aliased with any existing parameters, we create a new extended parameter to represent those values and record that in the parameter mapping. Section 3.2 describes the situations where the initial values are aliased with one or more existing parameters.

When the iterative analysis encounters a procedure call, it needs to find the effects of the call on the points-to function.

If the call is through a pointer passed as an input to one or more of the PTFs on the call stack, we add the potential values of that pointer to the specifications of the input domains for those PTFs. For each potential callee procedure, we first check if any of its PTFs apply. This involves building up a parameter mapping and comparing the input aliases to those recorded for the PTF. If the input aliases and function pointer values match, we use the parameter mapping to translate the final pointer values recorded in the PTF back to the current context. Otherwise, if the inputs do not match any of the existing PTFs, we reanalyze the procedure to produce a new PTF.

3 Pointer Representations

Since C programs commonly access memory using type casts and other features to override the high-level types, our algorithm is based on a low-level representation of memory. This allows us to handle the problematic features of C in a straightforward manner. Instead of using types to identify locations that may contain pointers, we assume that any memory location could potentially hold a pointer. We also refer to locations within a block of memory by their positions, not by their field names. We define a new abstraction, the *location set*, to specify both a block of memory and a set of positions within that block. For the most part, this low-level representation provides results comparable to those based on high-level types. However, our conservative approach may occasionally lead to less accurate results. This is a price we are willing to pay in order to guarantee safe results for all input programs¹.

We divide memory into blocks of contiguous storage, whose positions relative to one another are undefined. A block of memory may be one of three kinds: a local variable, a locally allocated heap block, or an extended parameter. Global variables are treated as extended parameters. A special local variable represents the return value of a procedure. Note that the heap blocks are only those allocated within a procedure or its callees; heap blocks passed in from a calling context are considered to be extended parameters.

We distinguish heap blocks based on their allocation contexts, grouping together all the blocks allocated in each context. The minimal context information is simply the statement that creates the block. Including the call graph edges along which the new blocks are returned, eliminating duplicate edges in recursive cycles, can provide better precision for some programs [2]. While this scheme is a good starting point, we have found that for some programs it produces far more heap blocks than we would like. We are currently investigating techniques to merge heap blocks that are elements of the same recursive data structure in accordance with our goal to only distinguish complete data structures from one

¹We do still place a few restrictions on the inputs. For example, we currently assume that pointers are not written out to files and then read in and later dereferenced.

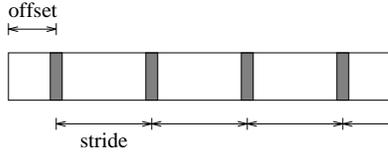


Figure 5: Members of a Location Set

another. For now, we limit the allocation contexts to only include the static allocation sites. That is sufficient to provide good precision for the programs we have analyzed so far.

3.1 Location Sets

Our goal with regard to aggregates is to distinguish between different fields within a structure but not the different elements of an array. That is, given an array of structures with fields x and y , the locations are partitioned into two sets, one containing all field x data and one containing all field y data. Our pointer analysis can be combined with data dependence tests to distinguish between different array elements.

We represent positions within a block of storage using location sets. A location set is a triple (b, f, s) where the base b is the name for a block of memory, f is an offset within that block, and s is the stride. That is, it represents the set of locations $\{f + is \mid i \in \mathcal{Z}\}$ within block b , as shown graphically in Figure 5. The offsets and strides are measured in bytes.

Expression	Location Set
scalar	(scalar, 0, 0)
struct.F	(struct, f, 0)
array	(array, 0, 0)
array[i]	(array, 0, s)
array[i].F	(array, f, s)
struct.F[i]	(struct, f % s, s)
*(&p + X)	(p, 0, 1)

Table 1: Location Set Examples
 f = offset of F and s = array element size

Table 1 shows the location sets that represent various expressions. A field in a structure is identified by its offset from the beginning of the structure. Except for array references, the stride is unused and is set to zero. A reference to an array element has a stride equal to the element size. If the elements of an array are structures, then the field information is captured by the offset. Since C does not provide array bounds checking, a reference to an array nested within a structure can access any field of the structure by using out-of-bounds array indices. Although such out-of-bounds references are rare and non-standard, we believe it is still important to handle them conservatively. Thus we treat an array nested in a structure as if it completely overlaps the entire structure.

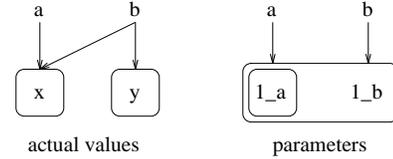


Figure 6: Subsuming Existing Parameters

This implies that the offset will always be less than the stride. We enforce that by always computing the offset modulo the stride whenever the stride is non-zero.

When the position of a location within a block is entirely unknown, we set the location set stride to one. This means that the location set includes all the locations within the block. This may occur due to pointer arithmetic. Although we recognize simple pointer increments, which are commonly used to address array elements, to determine the location set strides, we do not attempt to evaluate more complex pointer arithmetic. Instead, for each memory address input to an arithmetic expression, we add to the result a location set with the same base object but with the stride set to one. This conservatively approximates the results of any arithmetic expression. Because the positions of the blocks relative to one another are undefined, we need not worry about pointer arithmetic moving a pointer to a different block.

In summary, our location set representation has several advantages. Problems related to type casts and unions become irrelevant because we do not use the high-level types. It is also very easy to work with, especially when dealing with pointer arithmetic.

3.2 Extended Parameters

As discussed in Section 2.2, every object is represented by at most one extended parameter. When adding an entry to the initial points-to function, we first find the values of the pointer in the calling context. If the parameter mapping shows that an existing parameter already includes the same values, we reuse the old parameter instead of creating a new one.

For simplicity and efficiency, each initial points-to entry points to a single extended parameter. In cases where the initial values are aliased with more than one parameter, we create a new extended parameter that subsumes all the aliased parameters, and we replace all references to the subsumed parameters. Likewise, when the initial values are aliased with an existing parameter but also include new values, we subsume the aliased parameter with a new one. Figure 6 illustrates this. Parameter l_a is created first to represent the targets of a . When b is dereferenced later, we discover that its targets include the value represented by l_a but also another value. Thus, we create a new parameter l_b that replaces the old parameter l_a . This scheme reduces the number of extended parameters with some loss of precision. That is an acceptable trade-off for our purposes, but subsuming parameters is not essential to our algorithm and could easily

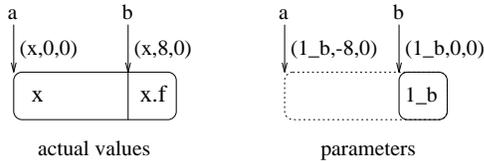


Figure 7: Using Negative Offsets for Structures

be omitted.

Aliases involving fields of structures can be a bit more complicated. When updating the initial points-to function, if we find a pointer to a field of an existing parameter, we can easily record that fact. However, if we encounter a pointer to a field before a pointer to the enclosing structure, we will create an extended parameter to represent the field, and then the other pointer will have to point *before* the existing parameter. Fortunately, our location sets solve this problem nicely. We simply allow the location set offsets to be negative, as shown in Figure 7. Emami solves this problem by always creating parameters for structures before any other parameters [5], but that cannot work here because we create the parameters as they are referenced.

3.3 Points-to Functions

Both the domains and ranges of the points-to functions are expressed in terms of location sets. At each statement in a program, a points-to function maps the location sets containing pointers to the locations that may be reached through those pointers. Thus, a points-to function is a mapping from location sets to sets of location sets.

It is important for the analysis to know which locations may contain pointers. Since we do not use the high-level types and since the points-to functions only contain entries for pointers that have already been referenced, we record separately for each block of memory the location sets that may hold pointers. Without that information, the analysis would have to conservatively evaluate every assignment as a potential pointer assignment. Although that would not affect the precision of the results (eventually the analysis finds which values may be pointers and removes any spurious results), we have found that it makes the analysis very inefficient.

For a variety of optimizations, it is important to know when a location *definitely* holds a certain pointer value. Within the pointer analysis itself, that information can enable *strong updates*, where assignments overwrite the previous contents of their destinations. Others have kept track of “possible” and “definite” points-to values separately [6], but that is unnecessary for our purposes. We only need that information at the point where a pointer is dereferenced. Since we assume that the input is legal, a location being dereferenced must contain a valid pointer. The points-to functions record all the valid pointers possibly contained in each location, and if there is only one possibility, it is safe to assume that is the value being dereferenced. Thus, we get the benefits of definite pointer

values without the overhead of tracking them separately.

4 Analyzing a Procedure

We use an iterative data-flow analysis to find the points-to functions within a procedure. In this section, we only discuss the process of analyzing procedures with no call statements.

```

EvalProc(proc *pr, PTF *ptf)
{
  /* iteratively analyze a procedure */
  do {
    changed = FALSE;
    foreach cfgNode nd in pr.flowGraph {
      if (no predecessors of nd evaluated)
        continue;
      if (nd is meet) EvalMeet(nd, ptf);
      if (nd is assign) EvalAssign(nd, ptf);
      if (nd is call) EvalCall(nd, ptf);
    }
  } while (changed);
}

```

Figure 8: Intraprocedural Algorithm

Figure 8 shows our data-flow algorithm. We simply iterate through all the nodes in a procedure’s flow graph until none of the points-to values change. We visit the nodes in reverse postorder, because it is important to know the input values before evaluating a node. This strategy is much simpler than a worklist algorithm, and it handles unstructured control flow with no extra effort.

4.1 Strong Updates

Strong updates, where assignments overwrite the previous contents of their destinations, are an important factor in the precision of pointer analysis. Unless we know that an assignment will definitely occur and that it assigns to a unique location, we must conservatively assume that that location potentially retains its old values.

Because strong updates make the node transfer functions non-monotonic, we introduce some constraints on the order in which the flow graph nodes are evaluated in order to guarantee that the algorithm terminates. Specifically, we never evaluate a node until one of its immediate predecessors has been evaluated, and we never evaluate an assignment until its destination locations are known [1].

We take advantage of the extended parameters to increase the opportunities for strong updates. A strong update is only possible if the destination of an assignment is a single location set representing a unique location. A location set is unique if it has no stride and the base represents a unique block of memory. The key is to recognize that an extended parameter representing the initial value of a unique pointer can be a unique block even if that pointer has many possible values in

the calling context. Since the pointer can only contain one of those possibilities at any one time, the extended parameter is a unique block within the scope of the procedure. Only when more than one location points to an extended parameter and the actual values for that parameter are not a single unique location must we mark the parameter as not unique. This greatly improves our ability to perform strong updates. Since a heap block represents all the storage allocated in a particular context, we assume that locally allocated heap blocks are never unique. On the other hand, local variables correspond directly to real memory locations so they are always unique blocks.

4.2 Sparse Representation

Because our analysis is interprocedural and needs to keep the entire input program in memory at once, we have gone to considerable effort to use storage space efficiently. Since we analyze heap data as well as global and stack variables, many possible memory locations could be included in the points-to functions. Fortunately, the information stored in the points-to functions is very sparse. Pointers typically have only a few possible values, so we record the possibilities using linked lists rather than bit vectors. Since the points-to functions usually do not change very much between two adjacent program points, we also incorporate the sparse representation described by Chase et al. [1]. This scheme only records the points-to values that change at each node.

Because of the sparse points-to function representation, looking up the values of a pointer requires searching back for the most recent assignment to that location. Beginning at the current node, we search back through the dominating flow graph nodes². If we reach the procedure entry when searching for an assignment to a formal or extended parameter, we compute the value of the initial points-to function for that parameter, if it has not already been recorded, as described in Section 3.2. This may add new extended parameters in the PTFs on the call stack.

Since we only search for assignments in the dominating nodes, each meet node must contain SSA ϕ -functions [3] to identify the values to be assigned in it. We insert these ϕ -functions dynamically as new locations are assigned [1]. The pseudo-code for handling a meet node is shown in Figure 9. For each ϕ -function, we look up the points-to values at each predecessor node and combine the results to get the new points-to values.

4.3 Evaluating Dereferences

Because location sets may overlap, more than one location set may refer to the same memory location. Values assigned to one location set must be observed by references to overlapping locations. Thus, when a pointer is dereferenced, we

²Instead of building “skeleton trees” [1], we just keep lists of assignments sorted according to a bottom-up traversal of the dominator tree.

```

EvalMeet(cfgNode *nd, PTF *ptf)
{
  /* iterate through the phi-functions */
  foreach locSet dst in nd.phiFuncs {
    locSetList srcs;
    /* combine values from each predecessor */
    foreach cfgNode pred in nd.preds {
      /* look up points-to values */
      srcs += lookup(ptf, dst, pred, NULL);
    }
    /* add points-to entry */
    assign(ptf, dst, srcs, nd);
  }
}

```

Figure 9: Evaluating Meet Nodes

iterate through all of the overlapping locations and look up in the current points-to function the values that have been assigned to each one. However, if the location being dereferenced is a unique location, values assigned to overlapping locations may have been overwritten by a strong update. In that case, we first find the position of the most recent strong update so that the lookup function will not look for assignments to overlapping locations prior to that point. Figure 10 summarizes this part of our algorithm.

```

EvalDeref(locSet *v, cfgNode *nd, PTF *ptf)
{
  locSetList result;
  cfgNode *strUpd = NULL;
  if (v is unique) {
    /* find the most recent strong update */
    strUpd = findStrongUpdate(ptf, v, nd);
  }
  /* find the locations containing pointers */
  locSetList locs = v.base.ptrLocations;
  foreach locSet loc in locs {
    if (loc overlaps v) {
      /* find values assigned to loc before
       node nd but not before strUpd */
      result += lookup(ptf, loc, nd, strUpd);
    }
  }
  return result;
}

```

Figure 10: Evaluating Dereferences

4.4 Evaluating Assignments

An assignment node in a flow graph specifies both the source and destination expressions, as well as the size of the value to be assigned. When building the flow graph from the intermediate representation of a program, we automatically convert the assignments to a “points-to” form. That is, since a variable reference on the right-hand side of an assignment refers

to the contents of that variable, we add an extra dereference to each expression on the right-hand side. Source and destination expressions may also involve pointer arithmetic. Simple increments are included in the strides of the location sets. We simply keep a list of all the constant location sets and dereference subexpressions found in other arithmetic expressions.

The process of evaluating an assignment begins by finding the locations identified by the source and destination expressions. To evaluate an expression, we iterate through its constant location sets and dereference subexpressions. Constant locations require no computation. Dereference expressions may include any number of nested dereferences, which we evaluate one level at a time. For each destination location, we then update the points-to function to include the values from all the potential source locations. Figure 11 shows this process for the case where the size of the assignment is one word or less.

```

EvalAssign(cfgNode *nd, PTF *ptf)
{
  locSetList dsts = EvalExpr(nd.dsts, nd, ptf);
  locSetList srcs = EvalExpr(nd.srcs, nd, ptf);
  foreach locSet dst in dsts {
    locSetList newSrcs = srcs;
    /* include the old values if this
       is not a strong update */
    if (dst is not unique)
      newSrcs += lookup(ptf, dst, nd, NULL);
    assign(ptf, dst, newSrcs, nd);
  }
}

```

Figure 11: Evaluating Assignments

In an aggregate assignment, where multiple words are assigned at once, all the pointer fields from the sources are copied to the destinations. If a source location contains a pointer value at an offset within the range being copied, we add that pointer value at the corresponding offset to the destination location.

5 Interprocedural Algorithm

We now describe how our algorithm handles procedure calls. To evaluate the effects of a call on the points-to function, we need to find a PTF that applies in the calling context. We can either reuse an existing PTF or create a new one. Figure 12 outlines this part of our algorithm.

5.1 Calls Through Pointers

The first step in evaluating procedure calls is to determine the target procedures. For most calls the target is a constant and this is a trivial step, but the target may also be specified by a function pointer. Fortunately, calls through pointers

```

EvalCall(cfgNode *nd, PTF *ptf)
{
  /* find & record function pointers values */
  procList targets = findCallTargets(nd, ptf);
  foreach proc pr in targets {
    paramMap map; recordActuals(nd, pr, map);
    PTF *tgtPTF;
    if (pr is not on the callStack) {
      tgtPTF = GetPTF(map, pr, nd, ptf);
      if (needVisit) {
        push (pr, tgtPTF, map) onto callStack;
        EvalProc(pr, tgtPTF);
        pop callStack;
      }
    } else {
      /* handle recursive call */
      tgtPTF = get PTF from callStack;
      /* add new aliases and func ptf values */
      updatePTFDomain(tgtPTF, map, nd, ptf);
      if (exit node of tgtPTF not reached)
        return; /* defer evaluation of nd */
    }
    ApplySummary(tgtPTF, map, nd, ptf);
  }
}

```

Figure 12: Evaluating Procedure Calls

are relatively easy to handle in pointer analysis, since the points-to functions record the possible pointer values.

Function pointer values may be expressed in terms of extended parameters. This is the one situation where we need to know the specific values represented by extended parameters. When an extended parameter is included as a potential target of a call, we check the parameter mappings up the call graph until we find the values that it represents. Since the function pointer values are part of the input domain specifications for the PTFs, we flag these extended parameters as function pointers and record their values in the PTFs.

5.2 Testing if a PTF Applies

The input domain of a PTF is specified by its initial points-to function and by the values of the parameters used as call targets. To test if a PTF applies, we check if these things are the same in the current context as they were when the PTF was created. We check the initial points-to function entries one at a time, in the order in which they were created. We then compare the values of the parameters that were used as call targets. If at any point they do not match, we give up and go on to the next PTF. The basic steps of this process are shown in Figure 13.

In the process of comparing the initial points-to function, we also build up a parameter mapping. If the PTF matches, we can then use this mapping to apply the PTF in the current context. For each entry in the initial points-to function, we find the actual values of the corresponding pointer in the current context and add the results to the parameter mapping.

```

GetPTF(paramMap *map, proc *pr,
      cfgNode *nd, PTF *ptf)
{
    PTF *home = NULL;
    /* check the existing PTFs */
    foreach PTF tgtPTF for pr {
        if (matchPTF(tgtPTF, map, nd, ptf)) {
            if (inputs have new pointer locations)
                needVisit = TRUE;
            return tgtPTF;
        }
        remove all extended parameters from map;
        /* check for the original context */
        if (tgtPTF.home == (nd,ptf)) home = tgtPTF;
    }
    needVisit = TRUE;
    if (home) {
        updatePTFDomain(home, map, nd, ptf);
        return home; /* reuse original PTF */
    }
    return allocatePTF(pr);
}

```

Figure 13: Finding an Applicable PTF

Just as when the PTF was created, this operation may create new initial points-to entries for the other PTFs on the call stack.

Sometimes the aliases and function pointer values for a PTF match, but we need to extend the PTF because new locations contain pointers. As described in Section 3.3, we record the location sets in each block of memory that may contain pointers. By allowing us to ignore operations that do not involve pointers, this makes the analysis more efficient. However, if an input location contains a pointer in the current calling context, whereas it did not when the PTF was created, the results in the PTF may not be complete. When that happens, we reanalyze the procedure to extend the PTF. Note that the resulting PTF will still be applicable to all the calling contexts in the original input domain, since assuming that more locations may contain pointers is only a matter of efficiency.

If none of the existing PTFs are applicable, we need to reanalyze the procedure. In general, we create an empty PTF and then revisit the procedure to compute its summary. However, this simple approach may waste a lot of storage. During the iterative analysis of a procedure, we may evaluate a call node several times with different input values on each iteration. If we create a new PTF for each set of inputs, we will likely end up with a lot of PTFs that only apply to intermediate results of the iteration. Our solution is to record the calling context where each PTF is created. If none of the existing PTFs match, but one of them was created in the current context during an earlier iteration, we update that PTF instead of creating a new one.

5.3 Applying a PTF

After we find an applicable PTF and a parameter mapping for that PTF in the current context, we translate the summary of the procedure back to the calling context. If the procedure call is through a pointer that has more than one potential value, we combine all the possible summaries and we do not perform any strong updates when applying them. The points-to function at the procedure exit summarizes the effects of the entire procedure, so we simply translate each points-to entry and add the result to the points-to function at the call site. Local variables do not exist in the calling context so we remove them when translating the points-to entries.

5.4 Recursive Calls

We use an iterative approach to handling recursive calls. Because of calls through pointers, we must identify the recursive cycles as the analysis proceeds. Since we already keep a stack to record the calling contexts, we can detect recursive calls by searching back to see if the call target is already on the stack. Instead of creating a new PTF for a recursive call, we just use the summary from the PTF that is already on the call stack. On the first iteration the summary may be empty, and we may need to defer evaluation of the recursive call. As long as there is some path that terminates the recursion, however, an approximate summary will eventually be provided. The iteration then continues until it reaches a fixpoint.

The PTF at the entry to a recursive cycle is an approximation for multiple calling contexts. We combine the aliases and function pointer values from each recursive call site with those recorded in the PTF. That may change the input domain for the PTF so that it no longer matches the original non-recursive calling context. To avoid that problem, we record two separate input domains for these recursive PTFs. One specifies the original input domain for the call from outside the recursive cycle, and the other combines the inputs from all the recursive calls.

6 Related Work

One of the most distinctive features of our algorithm is our conservative approach to handling all the features of the C language. Most of the previous work has made simplifying assumptions that rule out things such as pointer arithmetic, type casting, union types, out-of-bounds array references, and variable argument lists.

Our analysis is based on a points-to representation similar to the one described by Emami et al. [6]. Other work has used alias pairs. Choi et al. show how alias pairs can be compactly represented using a transitive reduction strategy [2]. In that compact form, the alias pairs are not much different than a points-to representation. There are some differences in precision between using full alias pairs and points-to functions, but neither is clearly superior [14]. We have found that

the points-to function is a compact representation that works well for analyzing C programs.

Our scheme for naming heap objects is taken directly from Choi et al. Most other work has used *k-limiting*, where some arbitrary limit is imposed on the length of pointer chains in recursive data structures [11]. Although *k-limiting* can sometimes provide more information, our algorithm is not intended to distinguish the elements of recursive data structures. That problem has been addressed by a number of others [1, 4, 8, 9, 13].

Using symbolic names to represent the values of pointers passed into a procedure was first suggested by Landi and Ryder [12]. They use “non-visible variables” to represent storage outside the scope of a procedure. Emami et al. use “invisible variables” for the same purpose. Our extended parameters are essentially the same as invisible variables, except that we choose to subsume parameters so that each initial points-to entry contains a single extended parameter.

Our work is most similar to the analysis developed by Emami et al. Their algorithm is based on an “invocation graph” which has a separate node for each procedure in each calling context. The size of an invocation graph is exponential in the depth of the call graph, implying that their algorithm is also exponential. Although our algorithm is still exponential in the worst case where every calling context has different aliases, it performs well for real input programs where the aliases are usually the same. We expect the precision of our results to be comparable to theirs, but there may be some differences. Our conservative handling of C occasionally causes us to propagate pointer values to more locations. This is the price we pay to get safe results for all inputs. Subsuming aliased parameters also gives up some precision to improve efficiency. On the other hand, our results may be better because we use the extended parameters to increase the number of strong updates.

Emami et al. mentioned the idea of using a memoization scheme similar to our algorithm. Our algorithm extends their design in several important ways. First, we parameterize references to global variables to increase the opportunities for reusing PTFs in different contexts. Second, we keep track of which pointers are actually referenced by a procedure. Thus, we avoid the overhead of creating and updating irrelevant parameters, and we prevent differing aliases among those irrelevant parameters from limiting the reuse.

7 Experimental Results

We have implemented our algorithm as part of the SUIF (Stanford University Intermediate Format) compiler system [16]. SUIF is a flexible infrastructure for compiler research. It includes a full ANSI C front end, so we are able to evaluate our analysis with large, realistic application programs. The SUIF system also includes many components that can benefit from points-to information. We have only just begun to take advantage of this. To illustrate the poten-

tial, we report some preliminary results of using the points-to information to parallelize numeric C programs.

We present preliminary results for a number of benchmarks, including several from the SPEC benchmark suites. Two of the floating-point programs from SPECfp92, *alvinn* and *ear*, are written in C and we include them both here. Of the SPECint92 integer programs, we present results for *compress* and *eqntott*. Most of the other SPECint92 benchmarks contain *setjmp/longjmp* calls or asynchronous signal handlers. We eventually plan to support *setjmp/longjmp* calls in a conservative fashion, but we do not know of any practical way to analyze programs that use asynchronous signal handlers, except for simple cases where the signal handlers exit from the programs. We have also analyzed some Unix utilities, *grep* and *diff*, and a number of the benchmarks used by Landi and Ryder [12].

Benchmark	Lines	Procedures	Analysis (seconds)	Avg. PTFs
allroots	188	6	0.18	1.00
alvinn	272	8	0.22	1.00
grep	430	9	0.65	1.00
diff	668	23	2.13	1.30
lex315	776	16	0.93	1.00
compress	1503	14	1.45	1.00
loader	1539	29	1.70	1.03
football	2354	57	6.70	1.02
compiler	2360	37	7.57	1.14
assembler	3361	51	5.82	1.08
eqntott	3454	60	9.88	1.33
ear	4284	68	2.99	1.13
simulator	4663	98	15.54	1.39

Table 2: Benchmark and Analysis Measurements

Since the focus of our work has been making pointer analysis efficient, we are primarily concerned with the times required to analyze the various benchmarks. Table 2 shows the benchmark programs sorted by size. The first two columns list the number of source lines and the number of procedures encountered during the analysis. The third column shows the analysis times in seconds for our algorithm running on a DECstation 5000/260. These times do not include the overhead for reading the procedures from the input files, building flow graphs, and computing dominance frontiers. Neither do they include the time required to write the results to the SUIF output files.

Our pointer analysis is clearly fast enough to be practical for these programs. In all cases, only a few seconds are required for the analysis. The amount of time can vary significantly, though, depending not only on the overall size of the input program but also on other characteristics of the program. For example, even though it is quite a bit larger than *eqntott*, *ear* is much easier to analyze. In general,

floating-point applications seem to be easy targets for pointer analysis. More complex programs will require somewhat more analysis time.

Besides the overall analysis times, we also measured some statistics to evaluate our use of PTFs. The results are very encouraging. The last column of Table 2 shows the average number of PTFs per procedure. These averages are all close to one. Moreover, most of the situations where a procedure has more than one PTF are only due to differences in the offsets and strides in the initial points-to functions. Combining PTFs in those situations could improve the efficiency with only a small loss in context-sensitivity.

The `compiler` program is an interesting case to examine in more detail. This program is a small compiler that uses a recursive descent parser. It has many procedure calls and a lot of them are recursive. Together these factors cause the invocation graph described by Emami et al. to blow up to more than 700,000 nodes [15]. This is for a program with only 37 procedures! For some larger applications, the exponential size of the invocation graph will be completely unreasonable to handle. Fortunately, our results show that it is unnecessary to reanalyze each invocation graph node.

Points-to-information is useful for many different compiler passes, but it is crucial for parallelization. Our first use of pointer analysis is to show that static analysis can be used to parallelize loops in numerical C programs. The current SUIF parallelizer uses our points-to-information to determine if formal parameters can be aliased. It then detects parallel loops and generates SPMD (Single Program Multiple Data) code for multiprocessors. It has many of the standard analyses for recognizing loop-level parallelism: constant propagation, induction variable recognition, and data dependence analysis. It also includes a few passes that are specific to parallelizing C programs: rewriting `while` loops as `for` loops where possible and rewriting pointer increments as array index calculations. After parallelization, the compiler generates a C output file which contains calls to our run-time library.

Program	Percent Parallel	Avg. Time Per Loop	Speedups	
			2 Proc.	4 Proc.
alvinn	97.7	7.4 ms	1.95	3.50
ear	85.8	0.2 ms	1.42	1.63

Table 3: Measurements of Parallelized Programs

We ran our parallelizer over `alvinn` and `ear`. We instrumented our run-time system to measure the sequential execution time spent in the parallelized portions of the code. This measurement is shown as a percentage in the first column of Table 3. We also measured the sequential time spent in each invocation of each parallelized loop to determine the granularity of the parallelism. The averages of these times are shown in the second column of Table 3.

For both programs, the compiler managed to parallelize

all the major loops. We ran the generated code on two and four processors of an SGI 4D/380. The speedups are shown in Table 3. The parallelized `alvinn` achieves very good speedups. The `ear` program performs reasonably well for two processors but it does not speed up much more with four processors. This is not surprising since there is so little computation in each parallelized loop and since the parallelized program suffers from a lot of false sharing. Here we have only shown the use of pointer analysis in a loop parallelizer for multiprocessors. The same pointer information could be used to generate parallel code for superscalar and VLIW processors, on which both `alvinn` and `ear` would perform very well.

8 Conclusion

We have presented a fully context-sensitive pointer analysis algorithm and have shown that it is very efficient for a set of C programs. This algorithm is based on the simple intuition that the aliases among the inputs to a procedure are the same in most calling contexts. Even though it is difficult to summarize the behavior of a procedure for all inputs, we can find partial transfer functions for the input aliases encountered in the program. This allows us to analyze a procedure once and reuse the results in many other contexts.

Even though our algorithm is still exponential in the worst case, we have so far found that it performs well. As long as most procedures are always called with the same alias patterns, our algorithm will continue to avoid exponential behavior. To be safe, after reaching some limit on the number of PTFs per procedure, we could easily generalize the PTFs instead of creating new ones.

Our analysis can handle all the features of the C language. We make conservative assumptions where necessary to ensure that our results are safe. Even though we may occasionally lose some precision due to these conservative assumptions, we believe it is important to handle the kinds of code found in real programs, even if they do not strictly conform to the ANSI standard.

Our success so far has been encouraging, and our next step is to experiment with large, real-world applications. Our preliminary evaluation suggests that our approach may scale well for larger inputs, since most procedures only require one PTF. We also need to use our pointer analysis in more compiler optimizations to determine if the results obtained are sufficiently precise. Although there is more work to be done, we believe this is a major step towards making pointer analysis practical.

Acknowledgements

We wish to thank Bill Landi for generously giving us his benchmark programs and Erik Ruf for providing some measurements of those programs. Both they and the reviewers

also gave us many helpful comments and suggestions. Many thanks to Jennifer Anderson, Shih-Wei Liao, Chris Wilson, and the rest of the SUIF group for their work on the SUIF parallelizer.

References

- [1] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.
- [2] J.-D. Choi, M. Burke, and P. Carini. Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, Jan. 1993.
- [3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Jan. 1989.
- [4] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.
- [5] M. Emami. A Practical Interprocedural Alias Analysis for an Optimizing/Parallelizing C Compiler. Master's thesis, School of Computer Science, McGill University, Aug. 1993.
- [6] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.
- [7] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, and M. S. Lam. Interprocedural Analysis for Parallelization: Preliminary Results. Technical Report CSL-TR-95-665, Computer Systems Lab, Stanford University, Stanford, CA 94305-4055, Apr. 1995.
- [8] W. L. Harrison III. The Interprocedural Analysis and Automatic Parallelization of Scheme Programs. *Lisp and Symbolic Computation*, 2(3):176–396, Oct. 1989.
- [9] L. J. Hendren and A. Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, Jan. 1990.
- [10] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, pages 244–251, June 1991.
- [11] N. Jones and S. Muchnick. Flow Analysis and Optimization of Lisp-like Structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice Hall, 1979.
- [12] W. Landi and B. G. Ryder. A Safe Approximate Algorithm for Interprocedural Pointer Aliasing. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.
- [13] J. R. Larus and P. N. Hilfinger. Detecting Conflicts Between Structure Accesses. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.
- [14] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. D. Choi, M. G. Burke, and P. Carini. Pointer-Induced Aliasing: A Clarification. *ACM SIGPLAN Notices*, 28(9):67–70, Sept. 1993.
- [15] E. Ruf. Personal communication, Oct. 1994.
- [16] R. P. Wilson et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices*, 29(12):31–37, Dec. 1994.