Interprocedural Pointer Analysis for C

John Lu

CRPC-TR96657-S
October 1996

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Interprocedural Pointer Analysis for C

John Lu
Rice University
Department of Computer Science

Abstract

Interprocedural pointer analysis provides information on the values pointer variables may have at various points in a program. It is especially useful in languages like C, where pointers are extensively used. This analysis does not improve code directly but is used by other optimizations.

This paper presents a framework in which to perform pointer analysis. Basic questions that any pointer analyzer must answer are discussed, and answers to these questions are developed. A working pointer analyzer is presented with performance results given for the analyzer. The effect of the analyzer on the running time of various programs is also shown.

1 Introduction

Many powerful optimization techniques rely on the availability of accurate information connecting the definitions and uses of values in a program. Examples of such transformations include dead code elimination, constant propagation, loop-invariant code motion, partial redundancy elimination, and global value numbering. All of these techniques have proved important in compilers for languages like Fortran. Experience suggests that these techniques are less effective when applied to code written in C. The combination of frequent procedure calls and pointer-based memory operations make it more difficult to determine precisely the relationship between the definitions and uses of values. In particular, pointer-based operations introduce fundamental ambiguities into the compiler's model of memory, while procedure calls further complicate the mapping of compile-time names to run-time storage locations.

Unless the compiler analyzes the behavior of pointer-based operations, it must assume that *any* pointer-based operation can access *any* memory location. Thus, any definition of a pointer-based value must be assumed to redefine every memory location and any use of a pointer-based value must be assumed to reference every memory location. This conservative set of assumptions introduces ambiguity into the results of any analysis of program behavior. Analysis of pointer-based operations can remove some of this ambiguity. Such analysis can improve optimization in two very different ways. First, better information about pointer-based operations will improve the overall quality of optimization by sharpening the information available to other analyses and transformations. Second, the compiler can use the results of pointer analysis as a direct basis for transforming the program. We are interested in pursuing both kinds of improvement.

Much work has already been devoted to developing practical pointer analysis methods for C. This document describes a proposed course of research on improved pointer analysis for C. It presents a new framework in which to perform the analysis. Section 2 examines the basic questions that any pointer analyzer for C must answer. Section 3 discusses previous work in the area and how it has answered these questions. Section 4 presents the new framework for pointer analysis in C, including an abstract, yet simple, model of how pointer analysis can be done. Performance results for the analyzer and programs it has analyzed are included. Section 5 lays out a program of further work.

2 Questions

Many different approaches have been attempted to analyze C. A useful way of categorizing pointer analysis is to consider the questions that the analyzer tries to answer.

This work has been supported by ARPA through Army Contract DABT63-95-C-0115 and by the State of Texas through Texas ATP award 003604-015.

```
subroutine caller()
   integer a, b
   call called(a,b)
   end

subroutine called(r,s)
   r = 1
   s = 2
   end
```

Fortran version

```
void caller() {
   int a, b;
   called(&a,&b);
}

void called(int *r,int *s) {
   *r = 1;
   *s = 2;
}
```

C version

Figure 1 Interactions between pointers and parameter binding

2.1 Choice of Problem

The first design decision for an interprocedural pointer analyzer is "Will it analyze pointers or aliases?" In alias analysis, the fundamental abstraction is the alias pair. An alias pair is simply a pair $\langle x, y \rangle$ where x and y are C expressions that may access the same memory location. The goal of alias analysis is to determine the set of alias pairs that can hold at a given point in a program.

Using alias pairs is an unfortunate consequence of thinking of pointer analysis in terms of the framework used to analyze aliases in FORTRAN caused by call-by-reference parameters[2]. Formal parameters in a call-by-reference language are not equivalent to memory locations. They are alternative names given to memory locations within a function. In this situation, alias pairs are the correct information to gather. In C, however, the situation is somewhat different. If we ignore unions, each memory location has a unique name, the name with which it is declared. Thus, the distinction between a name and a memory location is both artificial and unnecessary in C. However, even though a memory location has a unique name in C, it can be accessed through other names—the names of pointers that point to it. We can derive a relationship, called "points-to", for a C program that embodies this information. The points-to relationship maps a variable name into a set of memory locations. This more closely models the run-time situation in C; it appears to be the critical information to gather in analyzing the pointer behavior of C programs.

An added benefit of adopting the points-to abstraction is that it handles function pointers in an easy and natural way. The same machinery used to analyze memory pointers can track pointers to functions, as long as the compiler can recognize both the creation of such pointers and their actual use. Hendren noted this advantage in her 1994 PLDI paper [9].

2.2 Modeling Parameter Binding

The analyzer must include a mechanism for modeling the effects of executing a procedure call and a subsequent return on the state of the program's name space. The complications that arise from these events are a result of the parameter binding mechanism in C, which relies on call-by-value parameters [1]. This seems simpler than the call-by-reference convention used in Fortran; however, the presence of pointer variables in C re-introduces all of the problems caused by reference parameters in other languages, as Figure 1 shows.

In designing an analyzer, this problem raises another critical question: "How should the analyzer represent and name non-local memory?" Two solutions have been proposed in the literature: using abstract but representative names and using explicit names.

In a framework that uses representative names, the analyzer creates a new abstract name for the non-local memory that can be referenced at each call-site. These names are then "un-mapped" at the procedure's

return to reflect the effect of the procedure call on any non-local memory that it may have accessed. To see this more clearly, consider Figure 2. To model the memory locations that can be accessed by parameters `*p1` and `*p2`, the analyzer would create a single representative name, like `Rep1`. Only a single name is used because the analyzer must assume that `*p1` and `*p2` may point to the same location. (Using distinct representative names would exclude this possibility.) On exit from `called()`, the analyzer will know that `Rep1` can point to `globalA` and `globalB`. When it maps these results back into the call-site in `caller`, it will determine that `a` can point to either `globalA` or `globalB`. It will also discover that `b` can point to either `globalA` or `globalB`. The use of a single representative name has introduced imprecision into this very simple example.

A further difficulty with representative names arises in handling arbitrarily-sized data-structures. The analyzer must decide how many names to create for each pointer; the only real hint available is the type of the pointer. Unfortunately, programmers can (and do) create types that are both recursive and self-referential, like a graph. For a pointer to a graph, the analyzer cannot easily determine the number of representative names that must be created. This problem is analogous to modeling data structures in the heap.

When the code passes a pointer into a function, the pointer can point to an explicitly-named object. In this case, the analyzer should use the explicit name of the object rather than a representative name. In the previous example, using explicit names would produce different results. For example, the parameters are passed the names of distinct pointers, `a` and `b`. Using explicit names, the statement

```
*p1=&globalA;
```

would set the variable `a` declared in `caller` to point at `globalA`. No complicated mapping is needed. Furthermore, after the return, `a` and `b` point to `globalA` and `globalB` respectively. This is both correct and precise.

Clearly, an analyzer based on explicit names will have its efficiency limited by the number of names that it must instantiate. Thus, from an implementation perspective and an efficiency perspective, it is important to limit the number of non-local variables that the analyzer believes a function can access.[1] Otherwise, using explicit names will become too costly. Non-local names arise from three sources: global variables, heap-based variables, and local variables whose addresses are explicitly computed. The number of globals is finite. Unfortunately, the number of heap-based variables is unpredictable, in the general case, as is the number of concurrently live instances of a specific local variable.

```
void caller() {
  int *a,*b;
  called(&a,&b);
}

void called(int **p1, int** p2) {
  *p1=&globalA;
  *p2=&globalB;
}
```

Figure 2 Function modifies non-local memory

[1] Of course, the analyzer must represent all of the names that are possible. What we would like to do is limit the number of spurious names.

In analyzing a function that can access non-local names, care must be taken with addressed local variables of recursive functions (ALVRs). Since an ALVR can have more than one instance, the analyzer must be quite clear about what it means when it determines that an ALVR points to some name. Some approximation will be necessary. In our framework, we will approximate by having the single name for the ALVR represent all instantiations of the ALVR. If at a program point **p**, some instantiation of an ALVR, `r`, may point to a location `q`, then in our analysis at program point **p**, `r` should point to `q`. An important consequence of this approximation is that stores to an ALVR cannot "kill" old values. (*i.e.*, All pointer values in an ALVR before a store will still be in the ALVR after the store.) The non-addressed local variables of a recursive function need not be approximated in this way. When such a variable is accessed, only the most recent instance can be accessed. Thus, its name does not need to represent all instances, but only the topmost instantiation. See sections 4.1, 4.2 and 4.3 for a more detailed discussion of this issue.

The choice between using explicit names or representative names may require trade-offs between space, speed, accuracy, and simplicity. Explicit names should be more accurate, but more space will probably be required to give a function access to all the non-locals names it may use. Of course, creating representative names takes space, too. We can undoubtedly write programs where the representative names take more space than the explicit names, although we do not expect that to be the typical case. With explicit names no mapping and unmapping functions are necessary. This should simplify both the analysis and the analyzer. It is not clear which method will be faster, since using representatives will use fewer names, but it also requires mapping and unmapping functions. Section 4.6 presents performance results for my analyzer, which uses explicit names.

2.3 Program Representation

Another issue that affects the design of both the analyzer and the analysis algorithm is the design of the intermediate form used to represent the program. In particular, the level of abstraction in the representation determines, to a large extent, both the difficulty of deriving the initial information and the ease with which the results can be applied to the resulting code.

- A more abstract, source-level representation can simplify gathering initial information; in a lower-level, assembly-like representation, some analysis may be required to recreate information that was obvious in the source code. For example, an address expression may be implicit at source-level but be distributed across several basic blocks at assembly-level.

- A more abstract, source-level representation imposes the programmer's name space onto the results of analysis; in an assembly-level representation, each individual subexpression may have a unique name. For example, a source-level analysis is unlikely to disambiguate two pointer references that use induction variables introduced by operator strength reduction [6].

In designing an analyzer, the compiler writer should consider the uses to which the results will be put. We intend to use the results of our analysis to drive optimizations on a low-level representation of the program. Therefore, our analyzer will work from an assembly-like representation of the code. In general, we would expect the results of analysis on the low-level code to be of greater interest to low-level optimization.

2.4 Flow Sensitivity

In any interprocedural analyzer, the compiler writer must choose between a "flow-insensitive analysis" or a "flow-sensitive analysis". In general, the flow-insensitive algorithm will ignore control flow inside individual procedures, where the flow-sensitive algorithm will need to analyze the flow inside each procedure. For points-to information, the choice between flow insensitive and flow sensitive will change the results of the analysis significantly.[2]

[2]There are problems for which this is not true. May Modifies information is the classic example of a problem that gains no additional information from considering intraprocedural control flow.

Flow-sensitive analysis will, in practice, be slower [19]. We feel that the increased precision obtained with a flow-sensitive analysis is critical in computing points-to information. Thus, our analyzer uses a flow-sensitive algorithm.

2.5 Modeling the Heap

A common use of pointers in C programs is to track the addresses of data structures that have been dynamically allocated on the heap. Thus, much effort described in the literature has been directed at discovering and representing the shape of pointer-based structures constructed in the heap [8, 4, 17, 12].

From our perspective, the issue of understanding the internal structure of the heap is orthogonal to the rest of the analyzer.

- The simplest solution available to us would model the whole heap with a single name. All loads and stores involving the heap would refer to this name. Since the name represents multiple locations, stores to it cannot kill old values.
- An easy improvement to the simple model would split the heap by call-sites into the allocation routine. This would give a distinct name to the memory allocated by each call-site, improving the precision of the analysis at a small cost in complexity, speed, and space.
- Further refinement in heap modeling is possible. It may be possible to split the heap memory allocated by one call-site into separate regions, along either the time or space dimensions. For example, in the space dimension, we could split the memory generated by one call-site based on the structure of the memory allocated at the call-site. In the time dimension, we might try to generalize from splitting by call-sites to splitting by call-paths.

Splitting the heap by call-paths can be helpful. For example, in Figure 3, the analyzer could return two separate names for the two calls to `My_Malloc`. However, for memory allocators that recycle memory, this will not be useful. Since the same memory may be returned by different calls to the allocator, the analyzer would have to return the same heap name for all calls to the allocator. Another solution would be to allow the user to specify what calls allocate memory. This would accurately split the heap for all memory allocators.

Another popular approach to splitting the heap is *k-limiting*[13]. This approach allows up to k copies of a heap name to be generated where k is pre-determined. The goal of the *k-limited* approach is to allow for more accurate analysis of heap structures. If we have a heap allocated linked list, the first *k-1* nodes can be given unique names and the rest of the list can be approximated with the k-th name. This information is very difficult to generate. Furthermore, even if this information could be determined, it would not be very beneficial. In this analysis, a technique is useful only if it allows more precise determination of what memory locations loads and stores may access. Thus, splitting a heap name into two names, `@_heap1` and

```
main() {
  Node *a,*b;
  a=My_Malloc(sizeof(Node));
  b=My_Malloc(sizeof(Node));
}

Void *My_Malloc(Int size) {
  return malloc(size);
}
```

Figure 3 Splitting the heap by call-paths

`@_heap2` will be profitable only if memory operations that operate on only one of the two names can be found. Finding such operations is easy to do when the heap is split by call-sites. It may be very difficult when the heap is split by k-limiting.

2.6 Path Information

The final choice facing the compiler writer is how much context to record and use during the propagation of information. For each pointer, we can associate path information with it that describes the execution path or paths along which the pointer was created. The contextual information encoded in path information can eliminate two sources of inaccuracy. In Figure 4, if the analyzer lacks path information, it will conclude that `p1` can point to either `a1` or `a2`, and that `p2` points to the same values. This inaccuracy arises when the analyzer begins to analyze `called()`. It begins by merging the contextual information available about the various calling procedures. In this case, it merges the fact that `p` can point to `a1` with the fact that `p` can point to `a2`. Propagating these facts through the return and back into the assignment in the calling procedure results in the erroneous (but conservative) points-to information. This problem has been termed the realizable path problem.[15]

In Figure 5, `pp` points to `p1` and `p2` and `p` points to `a1` and `a2`. There are no inaccuracies with this information, but without path information it is impossible to know that `pp` and `p` cannot simultaneously point to `p1` and `a2` respectively or `p2` and `a1` respectively. Thus, in `caller1()` after the call to `called()`, `p1` will needlessly point to `a2`. I have named this problem the pointer mixing problem.

Although path information can eliminate both these inaccuracies, its use must be carefully considered. If full path information (*i.e.*, a listing containing all the call-sites in a path) is used, an exponential number of paths may need to be generated. Less powerful but more efficient approaches may be taken. The analyzer can arbitrarily limit the length of the path context that it retains, much as k-limiting places an arbitrary fixed bound on the length of models for heap-based data structures. All these approaches should be analyzed by considering their effect on the running time and space requirements of the pointer analyzer and the speedup they produce in analyzed programs.

```
void caller1() {
  int a1,*p1;
  p1=called(&a1);
}

void caller2() {
  int a2,*p2;
  p2=called(&a2);
}

int *called(int *p) {
  return(p);
}
```

Figure 4 Realizable path problem

```
void caller1() {
  int *p1,a1;
  called(&p1,&a1);
}

void caller2() {
  int *p2,a2;
  called(&p2,&a2);
}

void called(int** pp,int *p) {
  *pp=p;
}
```

Figure 5 Pointer mixing

3 Related Work

The literature contains a number of papers on pointer analysis that are related to our own work. We will consider work by five specific groups: Landi and Ryder[15, 16], Choi *et al.*[5], Hendren *et al.*[9], Wilson and Lam[22], and Ruf[20]. The following sections summarize their individual approaches and describe their position relative to the issues raised in the previous section.

3.1 Choice of Problem

Both Landi and Ryder, and Choi analyze alias pairs. Landi and Ryder simplify their analyzer by *k-limiting* the expressions that can appear in an alias pair. Choi notes that it is not necessary to record all of the alias pairs to perform alias analysis. A compact representation can be made by only maintaining the transitive reduction of the directed graph of alias relations. This is the clearest evidence that alias pairs are the wrong abstraction to use when analyzing C. For acyclic graphs, maintaining only the transitive reduction is equivalent to saying that only points-to information is needed. For cyclic graphs, points-to information may contain more information than a transitive reduction, but this information may be useful.[3] Hendren, Wilson and Lam, and Ruf use the points-to representation.

3.2 Modeling Parameter Binding

Landi and Ryder, Choi, and Hendren use representatives and mapping and unmapping functions to handle call-sites. They realize that this requires information about the type of the incoming pointer. They do not go into detail about how pointers to arbitrary-sized structures should be handled. Hendren notes a single memory location should not be represented by multiple names. She does not explain how to guarantee that incoming pointers do not point to the same location in order to allow multiple representatives to be used. If multiple representatives cannot be used, this could be a large source of approximations. Choi explicitly gives an example where two pointer arguments of the same type are initialized to point at different representatives. This will not work unless we guarantee that the two pointers cannot point to the same object. Wilson and

[3] Consider a graph with three nodes and all possible directed edges. The transitive reduction of this graph contains three edges, but the points-to representation contains all six. This extra information allows the points-to representation to determine that a single dereference from one node may reach either of the two other nodes. This information is lost if we keep only the transitive reduction.

Lam also use representatives to model parameter binding, but they are very careful about any aliasing that may be introduced by the initial pointer values to a function. They perform a separate analysis for every aliasing pattern that enters a function. Each analysis produces a partial transfer function which is applied to the appropriate call-sites to the function. Ruf uses explicit names to model parameter binding.

3.3 Program Representation

Landi and Ryder, and Hendren both use a C-like representation in their analysis. Wilson and Lam build their work in the SUIF (Stanford University Intermediate Format) environment[10]. It is not clear what kind of representation the other groups use.

3.4 Flow Sensitivity

Choi presents both flow-sensitive and flow-insensitive versions of his algorithm. He also notes that killing information can be used with a flow-insensitive version to improve its accuracy. Hendren, Landi and Ryder, Wilson and Lam, and Ruf all perform flow-sensitive analysis.

3.5 Modeling the Heap

Hendren has the simplest solution for analyzing heap memory. She creates a single name to represent all of it. This was not the focus of her research, and she notes that more sophisticated methods could be used. Landi and Ryder use C-expressions to name memory locations. Presumably this would include the heap. They do not say anything about special processing for expressions that may refer to heap locations. Choi is very aggressive in splitting the heap. A separate heap name is generated for each call-path. This is an extension of splitting the heap based on call-site. These names are further *k-limited*. Wilson and Lam split the heap by call-path. Ruf splits by call-site.

3.6 Path Information

Landi and Ryder compute alias information with the assumption of an initial alias holding on entry to the function. This conditional information allows alias information that is specific to a call-site to be calculated. Choi preserves context in his analysis by recording the call-site from which an alias was generated. Hendren's analysis is completely context specific, since her analysis is done over the invocation graph. She notes that theoretically this is exponential. Ruf's work examines the difference in accuracy between alias analysis with no path information and full path information. He does not develop a practical algorithm for analysis with full path information, since his goal is to determine the upper limit of what can be gained by using full path information. Wilson and Lam's work does not use path information, but should produce context specific results. This is done by applying partial transfer functions to simulate the effect of a call at a call-site. No imprecision is introduced from other call-sites, since a partial transfer function considers the context from each call to the function it models individually.

4 Current Work

In this section, we will describe our implementation of a pointer analyzer and examine its effectiveness. Our pointer analyzer uses pointers rather than alias pairs and explicit names rather than representatives. Our analyzer represents programs in an assembly-like language, performs flow-sensitive analysis, splits the heap by call-sites, and uses no path information. (See section 2 for a discussion of these choices.) This section contains five subsections. Subsection 4.1 describes an abstract pointer language, *PL*. Subsection 4.2 shows how pointer analysis can be performed on *PL*. Subsection 4.3 explains how the pointer behavior of a C program can be modeled with a *PL* program. Subsection 4.4 describes my implementation of a *PL* style pointer analyzer for C. Subsection 4.5 discusses the various sources of approximation in the analyzer.

	Type	Call-Sites	Heap	Path Information
LR	Alias	Representatives	*k-limiting*	one-level
CBC	Alias	Representatives	Call-Path w/ k-limiting	one-level
EGH	Points-to	Representatives	One Name	arbitrary-level
WL	Points-to	Representatives	Call-Path	arbitrary-level?
Ruf	Points-to	Explicit Names	Call-Site	none/arbitrary-level
Lu	Points-to	Explicit Names	Call-Site	none

LR Landi & Ryder
CBC Choi, Burke, & Carini
EGH Emami, Ghiya, & Hendren
WL Wilson & Lam

Figure 6 Comparison of approaches

4.1 *PL* Language

Here we define an abstract language *PL* that has the relevant features of pointer-based languages but is easy to analyze. A *PL* program consists of:

- global regions G_i
- functions F_j one of which, F_0, is the entry point into the program

A function F_j creates function regions $FR_{j,k}$ when it is invoked. These regions are destroyed on exit.
A *PL* function is defined as:

function	→ label **function** (register*) block*
block	→ label operation*[jump | branch | return]
operation	→ merge | load | store | address | map | call
jump	→ **JMP** label
branch	→ **BR** label label
return	→ **RTN** register
merge	→ **MR** register register*
load	→ **LD** register register
store	→ **ST** register register
address	→ **ADDR** [region | function] register
map	→ **MAP** register register arithmetic
call	→ [**CALL** label register register* | **CALLR** register register register*]
region	→ G_i | $FR_{j,k}$
arithmetic	→ A_l

Since *PL* is only concerned with pointer analysis, operations not relevant to pointer analysis have been left out. The branch operation is indeterminate, since we assume any intraprocedural path may be taken. The **MR** merge operation takes a value from a randomly selected source register and puts it in the first register. The **MAP** operation maps the address in the source register to a new address in the defined register. This map is defined by the arithmetic field A_l. To be a legal *PL* program, an instantiation of a function region $FR_{j,k}$ can only be referenced or modified by loads or stores if it is the topmost instantiation of $FR_{j,k}$.

This is not a natural consequence of the syntax, but is a restriction the programmer must guarantee. This requirement is necessary in order to make pointer analysis simpler. (See the explanation of how to propagate a region to a call-site in section 4.2) A consequence of this restriction is that ALVRs must be modeled by global regions. (See section 4.3)

4.2 Pointer Analysis

Our goal will be to label all loads and stores in PL programs with the regions they may reference or modify. A load or store labeled with a function region $FR_{j,k}$ indicates that the load or store may reference or modify the topmost instantiation of $FR_{j,k}$.

To label all the loads and stores, the program must be converted into SSA form[7]. This includes giving SSA names to all global regions and the topmost instantiation of all function regions. To generate SSA names for regions, the intermediate form must specify where a region may be modified and where it may be referenced. Loads, stores, calls, and returns have a list of regions, RR, that they may reference. Stores and calls also have a list of regions, MR, that they may modify. The RR and MR lists in a call operation do not include regions generated directly or indirectly by the call. In addition, all regions possibly referenced in a function, F_j, are given SSA names on entry to the function. This set is written as $Uses(F_j)$ and includes locals. These lists of regions should be as accurate as possible to reduce the time and space needed to build the SSA form of the program. Although the analyzer builds the SSA form for the whole program, this is not interprocedural SSA form. The analyzer just builds the intraprocedural SSA form of several functions at once. There are no phi-nodes connecting SSA names at call-sites with the parameters to which they flow. Interprocedural connections must be formed by other parts of the analyzer.

Every SSA name s will have a set of possible addresses it may contain written as $SET(s)$. The target of each **ADDR** operation is inserted into the worklist and initialized with the address of its region or function argument. The SSA name in $Uses(F_0)$ for each global region is initialized with any addresses the region may have when the program starts. If the region has an address, it is placed on the worklist. We remove items off the worklist and propagate them to their uses. The algorithm terminates when the worklist is empty. To propagate addresses, the analyzer uses the function **SetCopy**(dst, src) to set $SET(dst) = SET(dst) \cup SET(src)$. If this changes $SET(dst)$, dst is added to the worklist.

Figures 7 and 8 illustrate the actions needed to propagate an address in a load operation. In this example, we are propagating to a load operation with base register `r11` and target register `r12`. The memory regions that may be accessed by this load are listed in brackets after the operation. They are: `@_ap`, `@_bp`, and `@_cp`. These memory regions have been given SSA names, which are denoted by the subscripts. The "address" of these memory locations is shown above the box that represents them. Thus, memory region `@_ap`'s address is 0. Propagation to this load begins by examining the contents of the base register, `r11`, of the load. We start with the element 0. We look for any memory regions in the list of regions that this load may reference which are located at address 0. We see that $\texttt{@\_ap}_{13}$ represents `@_ap` at this point, which has the correct address. Since $\texttt{@\_ap}_{13}$ represents a memory region that may be pointed to by the base register of this load, we copy the contents of $\texttt{@\_ap}_{13}$, 90, and put it into the target register, `r12`. We follow an identical procedure with the other element, 4, in the base register and put the contents of $\texttt{@\_bp}_{14}$, 94, into `r12`. Since the address of memory location $\texttt{@\_cp}_{15}$ is not in `r12`, we do not copy its contents, 98, into `r12`. At the end of propagation the target register, `r12`, would contain 90 and 94. Since its contents have changed, `r12` would be added to the worklist.

The functions that perform propagation are shown in Figures 9 and 10. The steps needed to propagate addresses are fairly obvious except for (1) propagating a region to a store, (2) the treatment of regions at a call-site, and (3) propagating a function pointer to a call-site.

1. A region in the RR list for a store must have its addresses added to the corresponding region in the MR list because a store might not kill values in a region. Thus any addresses in a region before the store may still be in the region after the store. This is why stores need RR lists.

`LOAD r11 => r12` [$@\_ap_{13}$ $@\_bp_{14}$ $@\_cp_{15}$]

Memory Regions	0 @_ap	4 @_bp	8 @_cp

SSA Name	11	12	$@\_ap_{13}$	$@\_bp_{14}$	$@\_cp_{15}$
Original Name			@_ap	@_bp	@_cp
Contents	0,4		90	94	98

Figure 7 Before Propagation Step

`LOAD r11 => r12` [$@\_ap_{13}$ $@\_bp_{14}$ $@\_cp_{15}$]

Memory Regions	0 @_ap	4 @_bp	8 @_cp

SSA Name	11	12	$@\_ap_{13}$	$@\_bp_{14}$	$@\_cp_{15}$
Original Name			@_ap	@_bp	@_cp
Contents	0,4	90,94	90	94	98

Figure 8 After Propagation Step

2. There are two cases for handling a region r in the RR list at a call-site to function F_l. If r represents a region $FR_{l,n}$ that is created by F_l, then r is copied directly to the MR list at the call-site. Otherwise, r is copied into the corresponding region in $Uses(F_l)$. In the first case, this is done because r represents the topmost instantiation of $FR_{l,n}$ before the call-site. Within the call it is no longer the topmost instantiation. Thus, it cannot be referenced or modified, and the values in the topmost instantiation of $FR_{l,n}$ before and after the call are the same. This is reflected by copying r from the RR list to the MR list. In the second case, $FR_{l,n}$ is visible in the call, so its values must be copied into the call. If these values are modified by the call, the modified values will be placed in the call-site's MR list when the called function's **RTN** operation is processed. An **RTN** operation's RR list contains all non-local regions possibly modified by a call.

3. When a function pointer is propagated to a call-site, the calling-context must be propagated to the new target. In addition, the return context of the called function must be propagated to the call-site. This may seem unnecessary, because the return context will be generated by the newly propagated calling-context. However, if the newly propagated calling-context does not modify the existing return context (the called function may have other call-sites that may have already created a return context), the return context will not be properly returned.

Before analyzing the running time, it should be clear that this algorithm must halt. The algorithm continues only if an SSA name has a region added to its SET. Since there are a finite number of regions and SSA names, this process must terminate. Four parameters will be used to analyze the running time of the propagation algorithm.

- S Number of SSA names
- R Number of regions
- U Maximum number of uses of one SSA name
- D Maximum number of SSA names one operation can define

Each SSA name can be inserted and removed from the worklist at most R times. Thus, each SSA name will be processed at most R times. Each time an SSA name is processed, it may result in UD **SetCopy** calls.

```
Void PropagateToCall(Name ssaName,Operation callingOperation,Function calledFunction) {
    if (IsActualParameter(ssaName,callingOperation))
        SetCopy(CorrespondingFormal(ssaName,calledFunction),ssaName);
    else {
        /* propagating a region to a call */
        if (RegionIsLocalToFunction(ssaName,calledFunction))
            SetCopy(CorrespondingRegion(ssaName,callingOperation.ModifiedRegions),ssaName);
        else
            SetCopy(CorrespondingRegion(ssaName,calledFunction.UsesRegions),ssaName);
    }
}
```

Figure 9 Code for PropagateToCall

```
Void Propagate(Name ssaName,Operation operation,Function f) {
    /* propagating ssaName to a use in operation in function f */
    switch (operation.type) {
        MR     :  SetCopy(operation.definedRegister,ssaName);
                  break;
        LD     :  forAllRegions(region,operation.ReferencedRegions)
                      if (PointsTo(operation.BaseRegister,region))
                          SetCopy(operation.definedRegister,region);
                  break;
        ST     :  if (ssaName==operation.BaseRegister ||
                     ssaName==operation.ValueRegister)
                      forAllRegions(region,operation.ModifiedRegions)
                          if (PointsTo(operation.BaseRegister,region))
                              SetCopy(region,operation.ValueRegister);
                  else /* propagating a region to a store */
                      SetCopy(CorrespondingRegion(ssaName,operation.ModifiedRegions),ssaName);
                  break;
        MAP    :  SetCopy(operation.definedRegister,operation.arithmetic(operation.sourceRegister));
                  break;
        RTN    :  if (ssaName==operation.returnRegister)
                      forAllCallerSites(callOperation,f)
                          SetCopy(callOperation.definedRegister,ssaName);
                  else {
                      /* propagating a region to a return */
                      forAllCallerSites(callOperation,f)
                          SetCopy(CorrespondingRegion(ssaName,callOperation.ModifiedRegions),ssaName);
                  }
                  break;
        CALLR  :  if (ssaName==operation.functionPointerRegister) {
                      forAllCallees(calledFunction,operation) {
                          forAllParameters(parameter,operation)
                              PropagateToCall(parameter,operation,calledFunction);
                          forAllRegions(region,operation.ReferencedRegions)
                              PropagateToCall(region,operation,calledFunction);
                          forAllOutGoingRegions(region,calledFunction)
                              SetCopy(CorrespondingRegion(region,operation.ModifiedRegions),region);
                      }
                  } else {
                      forAllCalledFunctions(calledFunction,operation)
                          PropagateToCall(ssaName,operation,calledFunction);
                  }
                  break;
        CALL   :  PropagateToCall(ssaName,operation,operation.calledFunction);
                  break;
    }
}
```

Figure 10 Code for Propagate

SetCopy can be implemented to require O(#source addresses) + O(#target addresses) time or O(R) time. The only non-constant work needed to determine if a **SetCopy** operation needs to be done is in the **LD** and **ST** cases. In these cases, there is a test, PointsTo(operation.BaseRegister,region). If the addresses in operation.BaseRegister are represented by a linked-list, this can be done in O(R) time. Since this overhead is of the same order as the **SetCopy** operation, the worst case running time is O(R^2SUD). The worst case requires that every SSA name contain every address upon termination and that each **SetCopy** call adds at most one address. The average case appears to be much better than the worst case (see Figure 11). In the eleven programs tested in this paper, there were a total of 1,581,194 SSA names. Of these, 218,261 contained more than nine addresses. Over 72% of the SSA names had no addresses. The **indent** program was especially difficult to analyze because of a large array of structures that contained over 200 pointers. Since an array is approximated by one name in the analyzer, any access to this array would refer to all 200 pointers, greatly slowing down the analysis. Excluding **indent**, only 52045 SSA names contained over nine addresses and over 80% percent had no addresses.

4.3 Modeling and Analyzing C in *PL*

The pointer behavior of C programs can be modeled with the *PL* language. Global variables are modeled with global regions. Local variables that are not addressed or are not created by a recursive function can be modeled by a function region. ALVRs[4] must be modeled by a global region. These local variables cannot be modeled with function regions because we cannot guarantee that only the topmost instantiation will be referenced. An SSA name for a function region only contains what the topmost instantiation of that region may have at that point, which is not sufficient to model ALVRs. An SSA name for a global region representing an ALVR contains the possible addresses of any instantiation of that variable at that point in the program.

Program	Number of Regions	Number of SSA names	SSA names with X number of addresses			
			0-9	percent	0	percent
gzip	510	254017	254005	100.0	220202	86.7
cachesim	556	99981	99970	100.0	84709	84.7
water	277	13716	13714	100.0	11523	84.0
fft	85	4799	4797	100.0	3981	83.0
clean	836	199793	199768	100.0	155211	77.7
tsp	82	7351	7347	99.9	5788	78.7
dhrystone	71	2502	2500	99.9	1719	68.7
mlink	778	668308	666693	99.8	559929	83.8
allroots	51	978	976	99.8	807	82.5
bc	539	131231	80861	61.6	68581	52.3
indent	553	198518	32302	16.3	27481	13.8

Figure 11 *SET* sizes

[4]Of the programs tested, only **mlink** had an ALVR. ALVRs are easy to detect. A call-graph is built and all recursive functions are found. All addressed memory regions in these functions are ALVRs. It would be unwise to attempt to model ALVRs more accurately. Accurate modeling would produce little benefit, since ALVRs are rare. Also, any attempt to model them more precisely would require complex code to be written that would slow down the performance of the analyzer even if no ALVRs existed in the program. Separate addresses would have to be created for the multiple regions used to model an ALVR. Complicated code would be needed to adjust these addresses when they enter and leave the function in which they are created.

Since it represents multiple locations, addresses in these regions can never be killed. Fortunately ALVRs in C appear to be rare. Heap memory is also represented by global regions. Stores to heap memory cannot kill old values.

References to a region generate an **ADDR** operation. Accessing fields of non-heap structures can be modeled with a **MAP** operation. Accessing fields of a heap structure is not modeled with a **MAP** operation, since heap memory is modeled with a single name. The merge operation is used to model the effect of all other arithmetic operations on addresses (*e.g.* **MUL r1 r2 → r3** becomes **MR r3 r1 r2**). This is correct since, these arithmetic operations cannot generate addresses to a different region. Calls to heap generating functions can also be modeled with an **ADDR** operation. This splits the heap into sections based on the call-site.

PL could be extended to handle stores better. This was not done in order to show the core of *PL*. For regions modeling a scalar variable, we know that an explicit store to them will kill old values. This could be

```
struct inner {
  int a;
  int b;
};

struct outer {
 struct inner c[2];
 struct inner d;
};

main(int argc, char **argv) {
  int *a;
  struct outer g;

  a=(int *) malloc(sizeof(int));
  a=(int *) malloc(sizeof(int));
  f();
}

void f() {
  int *p,a;
  p=&a;
  f();
}
```

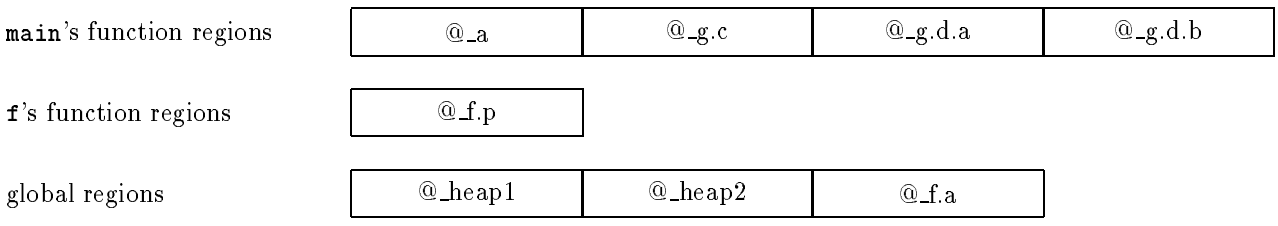

Figure 12 Program with eight regions

modeled in *PL* with a special store operation that did not require an *RR* list during analysis.

In Figure 12, we have a program with eight regions. Each non-structure field of a structure is considered a variable and given its own region name. If a field of a structure is itself a structure, (*e.g.*, `g.d` in function `main()`) then the naming is based on the fields of the interior structure. Note that variable `a` in function `f()` is an example of an ALVR and is given a global region.

4.4 Implementation

Our implementation was done within the Massively Scalar Compiler Project[3]. C-code is translated into `ILOC` our intermediate language, by C2I. C2I generates a region for each scalar and array variable. Each field of a structure and each compiler-generated location is also given a region. C2I also tells if a region represents a global variable or a local variable, if a region is ever addressed, and the region's size. C2I can explicitly annotate a load or store that does not use a pointer with the region it will reference or modify. C2I is also able to denote that some stores must kill old values in the region the store modifies.

`ali` collects all the files that C2I generates for a program. Each library function has a hand-made `ILOC` file that simulates its effect on pointers. Once all the files are parsed, `ali` generates the *MR* and *RR* lists for ambiguous loads and stores by inserting all addressed regions into these lists. The *MR* and *RR* lists for calls and returns and the list of regions possibly used by a function are limited by performing classical analysis.[5] Classical analysis is essential for keeping the SSA form of the analyzed program small. It can also improve the accuracy of pointer analysis. In Figure 13, `p` will contain the address of `g` before the call to `printf` in `f1`. Without classical analysis, this fact will enter the `printf` function, because all variables are in `printf`'s *RR* and *MR* lists. The variable `p` will be propagated to the return operation in `printf` and will be returned to all call-sites of `printf`. (Note this is an example of the false-return problem.) Thus, `p` will unnecessarily point to `g` immediately after the call to `printf` in `main`. With classical analysis, we know that `printf` cannot modify or reference `p`, so it will not appear in `printf`'s *RR* and *MR* lists.

```
int *p,g;

void f1() {
  p=&g;
  printf("hi");
}

main() {
  printf("hello");
  f1();
}
```

Figure 13 Mod and Ref analysis improves accuracy

[5] In classical analysis, we determine the memory regions a function can directly modify by looking at its loads and stores. Loads and stores through a pointer are conservatively assumed to access any addressed memory region. The total side-effect of a function is determined by processing each function in postorder. When a function is processed, the effect of its calls that do not correspond to backedges will already have been determined. Calls along backedges are conservatively assumed to access any global or addressed memory region. The call-graph used to perform the depth-first search is generated by assuming that pointer-based calls may call any addressed function.

The actual alias analysis starts by giving every region a numerical address. These addresses are chosen so that an add immediate generated to access a local in a stack frame can generate the correct address by adding its immediate to the address in the base register. The addresses of regions in a structure are also chosen so that the addressing arithmetic is preserved. Care is taken in handling add immediates to ensure that only addresses to the start of a region are generated. Otherwise, an infinite number of addresses might be generated by code that increments a pointer through an array. Using numerical addresses also makes initialization of the worklist easier. All references to a stack frame can be initialized by just inserting the base register onto the worklist.

Once the SSA names are initialized with these addresses, propagation is started. Propagation is done as described in the *PL* language section. Once alias analysis is finished, classical analysis is performed again to generate more accurate *RR* and *MR* lists for calls and return operations.

There is no provision for incremental compilation. However, separate compilation can be done with some user aid. Library functions are modeled with hand-crafted `ILOC` files that simulate their effect on pointers. The user can also create `ILOC` files to simulate portions of the program. If the user knows that a function `f()` has no side-effects, he can model it with an empty function. This way the rest of the program can be compiled without having to load in `f()` or any of the functions that can only be reached through it. This technique may be necessary for large programs.

4.5 Sources of Approximation

There are four sources of approximation in the analyzer:

- heap
- path information
- arrays
- ALVRs

The heap is approximated by one name for each call-site that generates heap memory. More sophisticated methods that split the heap along the space and time dimensions will be considered. No path information is used in the analyzer. This will be another area for future research. Arrays are treated as one unit. This is the main source of approximation when analyzing **indent** (See Figure 15). Names for ALVRs must represent all instantiations. This is the only approximation that recursion forces in the analysis. Recursion by itself will not cause any approximation in the analysis.

The first three sources of approximation appear in all algorithms for pointer analysis for C. Any solution for these three sources of approximation should be applicable in our framework as well. The fourth source of approximation is needed for the code shown in Figure 14. Suppose function `f()` was called with `f(0,NULL)`. This will result in recursive calls to `f()` to a depth of three. All three calls will have a local instantiation of `a`. The call to `f()` at depth zero has its instantiation of `a` addressed and passed to the other calls. The call at depth one stores through its instantiation of `a` to `global1`. The call at depth two sets the instantiation of `a` at level zero to point to `global2`. The analysis will unnecessarily conclude that the store through `a` at depth one at line `6` may modify `global2`. This occurs because `p` initially points to `a` due to the call on line `3`. Thus, line `8` will set `a` to point to `global2`. Upon exit from `f`, `a` will point to `global1` and `global2`. We unnecessarily conclude on line `6` that `a` may point to `global2`. This error occurs because all instantiations of `a` are represented by one name. It may be possible to get more accurate analysis by allowing multiple instantiations of a name (*e.g.*, `a_level0`, `a_level1`, `a_level2`, `a_level3`) in our analysis. However, this analysis would be very difficult. It would require knowing that certain portions of code only operate at a certain recursion depth. Furthermore, code for which such a technique would be useful is probably quite rare in C.

```
   int global1,global2;

   int f(int depth,int **p) {
     int *a;

1:   a=&global1;
2:   if (depth==0) {
3:     f(++depth,&a);
4:   } else if (depth==1) {
5:     f(++depth,p);
6:     *a=3;
7:   } else if (depth==2) {
8:     *p=&global2;
     }
   }
```

Figure 14 ALVR causes approximation

4.6 Results

The analyzer was tested on a small suite of programs. **clean** is the optimization pass of the Massively Scalar Compiler Project that removes empty basic blocks. **mlink** is a genetic linkage analysis program. **bc** implements a calculator language. **gzip** is the standard file compression and decompression program. **indent** transforms C programs into a pretty format. **cachesim** is a cache simulator. **water** is the numerical computation program from the SPLASH benchmarks. **fft** performs a Fast Fourier Transform. **tsp** solves the traveling salesman problem. **dhrystone** is the well-known benchmark program. **allroots** finds the roots of a polynomial equation.

Figure 15 shows the time and space needed to analyze the programs. The Lines column is the number of lines in all the source files used to make the program. This does not include header files. The number of lines for **clean** includes library files that contain functions not used in **clean**.

For each program three versions were prepared: one with no pointer analysis, one with classical analysis, and one with pointer analysis. By "no pointer analysis," we mean that ambiguous loads and stores, returns, and function calls were conservatively assumed to access any memory location.

These three versions were then optimized with: loop peeling, global value numbering, partial redundancy elimination, constant propagation, loop invariant code motion, and dead code elimination. The three optimized versions were then instrumented so that they would record the number of operations executed. The results show that pointer analysis can significantly speed up a program compared to a program with no analysis. For the programs tested, the average speedup was 7.6%. The results were not as good when compared to programs which had classical analysis. In this case, the average speedup was 1.8%. One interesting result is that pointer analysis can actually hurt performance. This was because the more precise information generated by the pointer analyzer allowed more loop invariant code to be moved. Unfortunately, this movement was not beneficial, since this code was in a location that was never executed.

The impact of pointer analysis is even more dramatic when we look at its impact on memory traffic. The unanalyzed version had 19.7% more stores than the version with pointer analysis. The classically analyzed version had 0.9% more.

The unanalyzed and classically analyzed versions had 24.7% and 7.0% more loads respectively than the

Program	Analysis Time (s)	Memory (M)	Lines
clean	13.9	40.9	11191
mlink	43.0	110.9	9264
bc	169.7	26.7	7583
gzip	8.2	28.0	7331
indent	674.4	46.9	5955
cachesim	4.2	16.4	2849
water	0.4	2.7	1345
fft	0.2	1.3	1037
tsp	0.3	1.2	760
dhrystone	0.1	0.7	534
allroots	0.1	0.3	215

Figure 15 Analyzer Statistics

Program	Original	Modref	**ali**	% Speedup over	
				Original	Modref
water	14672886	13554931	12795448	14.7	6.0
dhrystone	650195	588176	558176	16.5	5.4
tsp	728990	708337	679273	7.3	4.3
bc	5821567	5510190	5416170	7.5	1.7
mlink	145204328	137088645	134937328	7.6	1.6
allroots	1069	1047	1037	3.1	1.0
clean	1155893	1111025	1102972	4.8	0.7
fft	14069183	13125869	13065189	7.7	0.5
gzip (enc)	5834574	5808365	5776916	1.0	0.5
gzip (dec)	991063	988552	988168	0.3	0.0
cachesim	13472239	11974137	11972515	12.5	0.0
indent	896766	827127	827907	8.3	-0.1

Figure 16 Results

Program	Original	Modref	**ali**	% Speedup over	
				Original	Modref
dhrystone	82007	60007	56007	46.4	7.1
fft	1055908	1036664	1016176	3.9	2.0
gzip (enc)	279882	279869	274715	1.9	1.9
clean	79932	76387	76385	4.6	0.0
mlink	6090638	5490006	5489241	11.0	0.0
cachesim	1246105	594474	594474	109.6	0.0
water	1034219	1034219	1034219	0.0	0.0
tsp	51049	51047	51047	0.0	0.0
gzip (dec)	17786	17562	17562	1.3	0.0
indent	68855	67404	67404	2.2	0.0
allroots	13	11	11	18.2	0.0
bc	365517	265747	265747	37.5	0.0

Figure 17 Stores

version with pointer analysis. This is especially encouraging since loads can be very expensive operations. Since microprocessor speed is increasing faster than memory speed, these memory traffic improvements will become more and more important.

5 Future Work

Several issues should be explored further. Two of these areas, heap modeling and path information, have already been mentioned in Section 2.

1. Model the heap more accurately.
2. Explore the use of path information for more precise analysis.
3. Examine the effect of unions and type casts on our analysis.
4. Find transformations that can directly use the information to improve pointer-based codes.

5.1 Heap-Splitting

Currently, the analyzer models the heap by making separate names for the heap memory generated by each `malloc()` call-site. We plan on further refining our model of the heap by the methods (*i.e.*, splitting along call-paths, user-specification of memory allocators, and splitting by fields of heap allocated structures) mentioned in Section 2.

When refining the heap model, it should not be arbitrarily split. Generating heap regions is an expensive operation in the analyzer. Heap regions are addressed regions, so they will appear on the region lists of all calls and pointer-based memory operations. This will expand the SSA name space and require more memory. Modeling the heap more accurately is not the ultimate goal. It may or may not produce more accurate pointer analysis. This in turn may or may not result in performance increases in the analyzed code. We must weigh all the potential improvements against the increased time and space required. This evaluation will be done by running various versions of the analyzer on the test suite.

Program	Original	Modref	**ali**	% Speedup over	
				Original	Modref
water	3198626	2533613	1784412	79.2	42.0
dhrystone	74033	64018	54018	37.0	18.5
tsp	135071	124636	107001	26.3	16.4
mlink	33219484	27723944	26264447	26.4	5.6
bc	936994	820604	792242	18.2	3.6
fft	1924513	1253023	1216671	58.2	3.0
gzip (enc)	961509	915398	889141	8.1	3.0
allroots	163	145	141	15.6	2.9
clean	199961	175789	171927	16.3	2.2
gzip (dec)	138307	136145	135802	1.8	0.2
cachesim	2103377	1901788	1900867	10.6	0.0
indent	194601	144099	144877	34.2	-0.5

Figure 18 Loads

5.2 Path information

The results in this paper were produced without using path information. We plan to extend our analyzer to use the various types of path information introduced in Section 2. All implementations of path information will be evaluated by testing their performance on our test suite. We will consider both the effectiveness of path information to improve code and the extra costs needed to analyze with path information.

5.3 Safety

The analyzer assumes that it can identify where addresses to new regions are generated. In the analyzer, the only place where these addresses should be generated are:

- load immediate of the address of a global
- add to a stack pointer
- add to a structure pointer

We are able to distinguish between an add operation that generates an address to a new region and an add to an array pointer by the layout of memory regions in our analyzer's address space. The code in Figure 19 will illustrate how this works and a problem in this approach. The regions for this code can be seen at the bottom of the Figure. (Note that `p` does not have a region since it is allocated to a register) Regions are shown with their address above their top left corner. The code in this first section would produce the `ILOC` in Figure 20. Register `r0` will be initialized with address 0, since this is the base of the stack frame. Thus, we recognize that the `iADDI` generates an address to a new region because the resulting address, 4, is the start of region `@_c.b`. On the other hand, in the `iADD` operation, `r4` will contain the address 4, and `r5` will contain 4. When these are added together, the result, 8, is not the base of any region, so we do not add this address to `r6`. Instead, we assume that 4 is the address of an array region, so when we add to it we generate the same address, 4. Thus, 4 will be inserted in the list of addresses in r6.

Problems with this process can occur when we use type casts and unions. For the code in the second section of Figure 19, the analyzer will incorrectly conclude that the first store through `p` stores to `@_c.b`, and the second store through `p` stores to `@_c.a`. This occurs because the analyzer does not understand how

```
struct foo {
  int a,b[2];
};

void f() {
  int        *p;
  struct foo c;

/* good addressing */

  p=c.b;
  p[1]=3;

/* bad addressing */

  p=(int *) &c;
  p[1]=3;         /* analyzer thinks this stores to c.b */
  p+=2;
  p[0]=3;         /* analyzer thinks this stores to c.a */
}
```

f's function regions

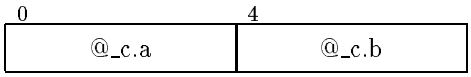

Figure 19 Good and bad address creation

```
        9       iADDI   4 r0 => r3              # get address of f_c
        9       i2i     r3 => r2
        10      i2i     r2 => r4
        10      iLDI    4 => r5
        10      iADD    r4 r5 => r6
        10      iLDI    3 => r7
        10      iSTor   ! 4 0 r6 r7
```

Figure 20 `ILOC` code with no address creation problems

to deal with type casts. All stores through `p` should be to the same region. Unfortunately, the underlying structure that was inherited through the type cast puts a region, `@_c.b`, in the middle of the array pointed to by `p`. The first store happens to hit the base of this region, so we conclude that it is a store to `@_c.b`. When `p` is incremented, this lands in the middle of region `@_c.b`. Since this is not the base of a region, the analyzer assumes this is an array increment and sets `p` to its original address. Thus, `p` points to region `@_c.a` after the increment. Further work needs to be done to ensure that this case is handled correctly. More generally, we need to answer how unions and type casts will affect our analysis.

5.4 Transformations

We will also look for code transformations that directly improve code using the implemented points-to analysis. For example, we plan on implementing a pass to perform register promotion. In register promotion, variables that are allocated in memory are promoted to registers. This can be done in the regions where the variable is never accessed through a pointer.

6 Acknowledgments

We wish to thank Philip Schielke for implementing the C2I front end, and Tim Harvey for infrastructure support.

We also thank Alejandro A. Schäffer for allowing us to test **mlink**. **mlink** comes from version 2.2 of the FASTLINK[14, 21] implementation of the LINKAGE[18] package for genetic linkage analysis. Genetic linkage analysis is a statistical technique used to map genes and find the approximate location of disease genes. LINKAGE/FASTLINK is a package of programs used by geneticists around the world to find the approximate location of disease genes. In our testing, **mlink** was run with the CLP data set. The CLP data set describes several families with many members who have autosomal dominant nonsyndromic cleft lip and palate[11]. We thank Dr. Jacqueline T. Hecht for contributing disease family data. Development of the CLP data set was supported by grants from the National Institutes of Health and Shriners Hospital.

Finally, we wish to thank Bill Landi for the **allroots** program.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

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

[3] Preston Briggs. The massively scalar compiler project. Technical report, Rice University, 1994.

[4] D.R. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. *SIGPLAN Notices*, 25(6):296–310, June 1990.

[5] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, 19(6):232–245, January 1993.

[6] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11), November 1977.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[8] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994.

[9] Maryam Emami, Rakesh Ghiya, and Laurie Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *SIGPLAN Notices*, 29(6):242–256, June 1994.

[10] R.P. Wilson *et al.* SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[11] J. T. Hecht, Y. Wang, B. Connor, S. H. Blanton, and S. P. Daiger. Nonsyndromic cleft lip and palate: No evidence of linkage to hla of factor 13a. *American Journal of Human Genetics*, 52:1230–1233, 1993.

[12] Joseph Hummel, Laurie J. Hendren, and Alexander Nicolau. A general data dependence test for dynamic, pointer-based data structures. *SIGPLAN Notices*, 29(6):218–229, June 1994.

[13] N.D. Jones and S.S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, 8(6):66–74, January 1982.

[14] Robert W. Cottingham Jr., Ramana M. Idury, and Alejandro A. Schäffer. Faster sequential genetic linkage computations. *American Journal of Human Genetics*, 53:252–263, 1993.

[15] W. Landi and B.G. Ryder. Pointer-induced aliasing: A problem classification. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

[16] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, June 1992.

[17] J.R. Larus and P.N. Hlifinger. Detecting conflicts between stucture accesses. *SIGPLAN Notices*, 23(6):21–34, June 1988.

[18] G. M. Lathrop, J. M. Lalouel, C. Julier, and J. Ott. Strategies for multilocus analysis in humans. *Proceedings of the National Academy of Sciences of the U.S.A.*, 81:3443–3446, 1984.

[19] Eugene W. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 219–230, Williamsburg, Virginia, January 1981.

[20] Erik Ruf. Context-insensitive alias analysis reconsidered. *SIGPLAN Notices*, 30(6):13–22, June 1995.

[21] Alejandro A. Schäffer, Sandeep K. Gupta, K. Shriram, and Robert W. Cottingham Jr. Avoiding recomputation in linkage analysis. *Human Heredity*, 44:225–237, 1994.

[22] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Notices*, 30(6):1–12, June 1995.