ALPHA: A Dependence-based Intermediate Representation for an Optimizing/Parallelizing C Compiler (Submitted for publication)

Laurie J. Hendren
Guang R. Gao
Vugranam Sreedhar

School of Computer Science, McGill University
318 McConnell Engineering Building
3480 University Street
Montreal, Quebec, CANADA
H3A 2A7
{hendren,gao,sreedhar}@cs.mcgill.ca

To face the challenges of compiling diverse and complex C programs, it is increasingly important to provide effective optimizing and parallelizing compilers that can handle analyses and transformations of programs that use a wide variety of data structures, including pointers. An integral part of this development is the design and implementation of intermediate representations that accurately and efficiently capture aliases and dependences. Our approach to this problem, is the development of ALPHA, a dependence-based representation, built upon our previous structured representation, SIMPLE. The ALPHA representation shares many of the advantages of the Static Single Assignment (SSA) intermediate form, but it has the further advantage of naturally handling dependences due to pointers. This makes our approach applicable to languages like C and Fortran90. We have implemented our approach in the McCAT C Compiler and we present our initial experience and measurements based on this implementation.

Keywords: Structured intermedidate representations, sparse representations, parallelizing/optimizing compilers, alias analysis, points-to analysis, dependence analysis

1 Introduction and Motivation

The effective compilation of many important C programs, including some of the so-called "grand challenge" problems, requires sophisticated compile-time analyses and optimizing/parallelizing transformations. The choice of intermediate representation greatly impacts the development of analysis methods. For example, the recent development of representations like the SSA form [17], and *sparse data flow evaluation graphs* [14], has led to the development of efficient analyses that use these forms [36, 19, 9, 42]. The main motivation of our work was to investigate and develop an intermediate representation that would allow us to support a wide variety of analyses and optimizing/parallelizing transformations in a straightforward and efficient manner. On one hand, one requires a representation that lends itself to the development and application of accurate alias and dependence analysis.[1] On the other hand, one would like dependence-based representations that support a wide-variety of efficient (sparse) analyses including those that have been successfully developed for the SSA form and other sparse representations.

An important facet of our work is that we wanted intermediate representations that can handle imperative programs including programs that use pointer data structures. Thus, we require both an intermediate representation that supports alias analysis in the presence of pointers, and a dependence-based intermediate representation that can represent dependences due to pointer aliasing in a natural fashion.

The development of our representations has been motivated by our experiences in building the McCAT C compiler, an optimizing/parallelizing compiler targeted at high-performance architectures that support fine-grain parallelism and/or multi-threading [24]. Figure 1 shows both the overall structure of the compiler, and the new additions proposed in this paper (as indicated by the inset). The high-level structure, as shown at the top of the figure, makes use of three structured intermediate representations FIRST, SIMPLE, and LAST [24]. Each representation is a structured AST that fulfills a specific role in the compiler. The FIRST representation provides a clean separation between the front-end and the back-end of the compiler, the SIMPLE representation provides a structured and simplified program representation that is convenient for alias and dependence analysis, and the LAST representation exposes low-level details that are appropriate for register allocation and instruction scheduling [28].

The accuracy and the correctness of the alias analysis, especially for heap-allocated objects, is very sensitive on how the flow of the analysis proceeds, and on the choice of the intermediate representation used for performing the alias analysis. We have carefully designed our intermediate representation SIMPLE for performing accurate and correct alias analysis. Our analysis method is based on a compositional analysis, for which we need structured programs. To this end, the McCAT compiler includes a restructuring phase that removes all goto statements from C programs. Thus, all of our analyses in McCAT compiler are performed on structured intermediate representation.

In our previous work, we have presented the design and implementation of SIMPLE and LAST. In this paper we focus on the development of the ALPHA dependence-based representation. As illustrated in the lower part of Figure 1, we start with the SIMPLE representation, and through various analyses and transformations we create the skeleton of the α-TREE , and then finally the α-TREE with dependences.

The main contributions of this paper are:

- We briefly summarize the foundations upon which we build: (1) the SIMPLE intermediate representation, and (2) the interprocedural *points-to analysis* that is used to compute accurate dependence information even in the presence of stack-based pointers. The SIMPLE representation and the points-to analysis makes use of the concept that every important location should have a name. This feature

[1] In fact, it is these analyses that allow one to build the dependence-based representations.

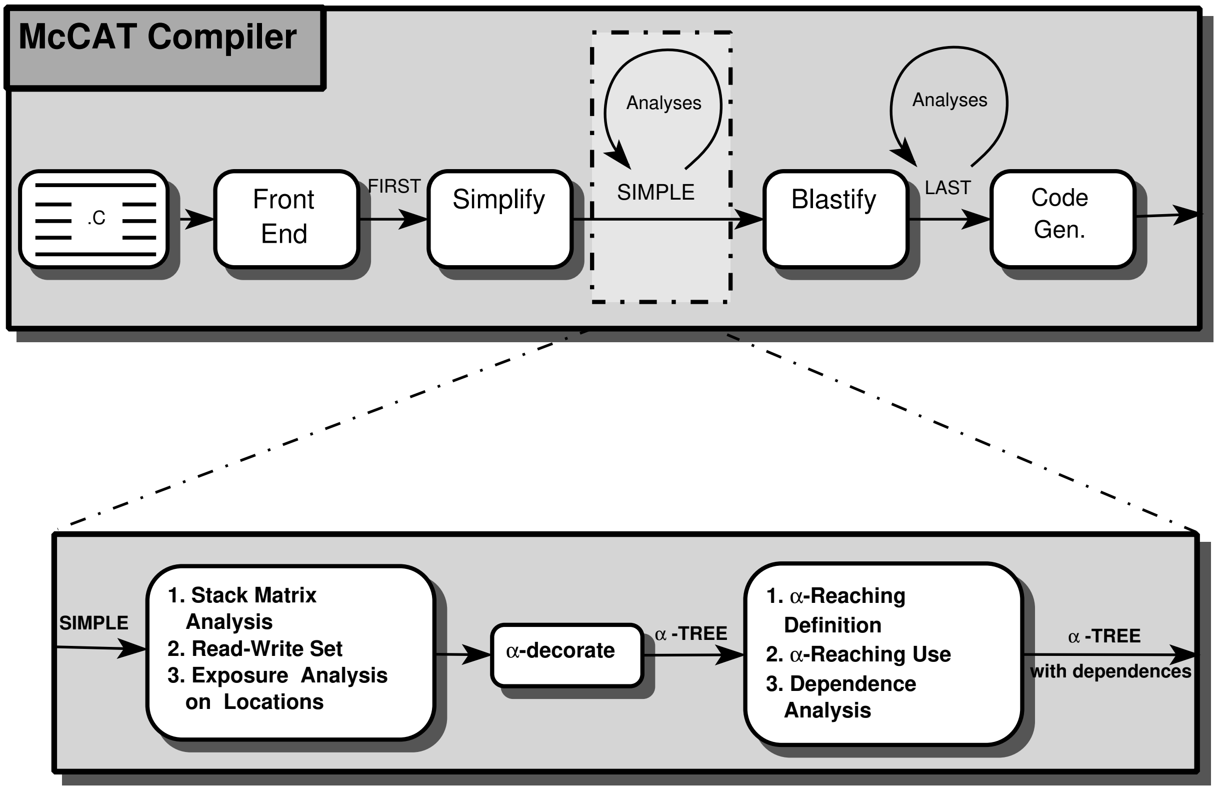


Figure 1: The McCAT Compiler

enables us to build the dependence-based representations in a straightforward manner.

- We introduce a new type of structured intermediate representation, α-TREEs and we demonstrate how this representation can be efficiently constructed from SIMPLE. The α-TREE representation differs from SIMPLE in that it includes α-statements that capture dependences entering and exiting structured control regions corresponding to conditionals and loops. This intermediate form is efficiently and correctly constructed even in the presence of pointers, `break` and `continue` statements in loops, and recursion. Furthermore, we show how the α-TREE representation lends itself to the development of efficient analyses. In particular, we show how reaching definitions can be efficiently implemented in both space and time. We also discuss other applications of α-TREEs for sparse analyses.

The remainder of this paper is structured as follows. The next section, Section 2, provides a short introduction to both the SIMPLE intermediate form, and the alias analysis method we use as the foundation for further analysis and transformation. In Section 3 we discuss the process that is used to create the α-TREE representation. In section 4 we discuss our implementation and give some experimental measurements. In Section 5 we provide an overview the potential uses of the α-TREE representation. Finally, we discuss how our work relates to other intermediate representations in Section 6, and give our conclusions in Section 7.

2 SIMPLE and Points-to Analysis: A Summary

As we stated in the introduction, one of our major goals was to develop an intermediate representation that can deal naturally with imperative programs and pointers. In order to achieve this goal we have designed both an appropriate structured intermediate form - SIMPLE, and an interprocedural alias analysis method called *points-to analysis*[24, 20]. Both SIMPLE and *points-to analysis* form the foundations for the development of our new α-TREE representation. In this section we present a summary of SIMPLE representation and points-to analysis. For a detailed description, interested reader are encouraged to see [24, 20, 40].

2.1 SIMPLE

The SIMPLE intermediate representation has been designed specifically to provide a good basis for interprocedural alias analysis. The key features of SIMPLE are:

- The simplified program is composed of simple assignment statements, and structured compositional statements (`while`, `do-while`, `for`, `if-else`, and `switch-case`). The simplified program has structured control flow except for the explicit use of `break` and `continue` in loops, and `return` to exit from procedures. All of the compositional statements are simplified to conform to simple rules. For example, switch statements are transformed so that each alternative branch ends with a `break`, `continue`, or `return`. Furthermore, any expression in the loops or conditional constructs are transformed so that all side-effects occur in their bodies, and thus expressions are guaranteed to be side-effect free.
- Variable references are simplified to one of: a scalar reference, a structure reference, an array reference, or a one-level pointer reference. In addition, all type information is available and all type-casting is made explicit.

Two examples of simplified programs are given in Figure 2(b) and Figure 3(b). Further details about SIMPLE may be found elsewhere [24, 40]

2.2 Points-to Analysis

The fundamental point of our approach is that rather than collecting alias information in the form of alias pairs [12, 11, 13, 15, 16, 32, 33], we have chosen to directly abstract relationships between locations. More specifically, we use an abstraction which associates a name with every important stack location, and we use abstract interpretation techniques to approximate the *relationships* between these abstract locations.[2]

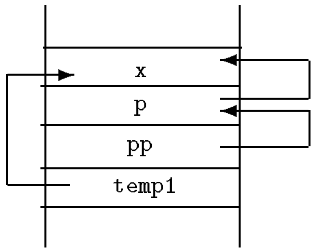
The goal of our analysis is to: (1) compute all the relationships between the important abstract stack locations, and (2) use the information about relationships to compute read and write sets for each statement. Let us begin by illustrating the *points-to* problem with a very simple example. Consider the program in Figure 2(a), and its SIMPLE form in Figure 2(b). Note that the statement `**p = 2` has been converted into two statements `temp_1 = *p; *temp_1 = 2`. By simplifying each reference to one level of indirection, we provide a name for each location on the stack.

```
{ int x, *p, **pp;

  x = 3;
  p = &x;
  pp = &p;
  **pp = 2;
}
```

(a) Example program

```
{ int x, *p, **pp;
  int *temp_1;

  x = 3;            /* 1 */
  p = &x;           /* 2 */
  pp = &p;          /* 3 */
  temp_1 = *pp;     /* 4 */
  *temp_1 = 2;      /* 5 */
}
```

(b) SIMPLE representation



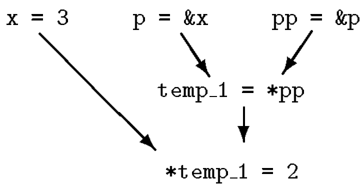
(c) Abstract Stack

	x	p	pp	temp1
x				
p	*D*			
pp		*D*		
temp1	*D*			

(d) Points-to Matrix

S	Write(S)	Read(S)
`x = 3`	{x}	ϕ
`p = &x`	{p}	ϕ
`pp = &p`	{pp}	ϕ
`temp_1 = *pp`	{temp_1}	{pp, p}
`*temp_1 = 2`	{x}	{temp_1}

(e) Read/Write Sets



(f) Dependencies

Figure 2: A simple example of SIMPLE and *points-to analysis*

Given the simplified representation, we perform the first step of the analysis by computing for each program point the "points-to" relationships between important locations on the stack. Figure 2(c) gives a pictorial view of the relationships between the abstract locations at program point 4 for the simplified program given in Figure 2(b). In this example, we see that "`pp` points to `p`", "`p` points to `x`", and "`temp_1` points to `x`". Figure 2(d) gives the concrete *points-to* representation actually used in the analysis. In general, each entry in a points-to matrix is one of: (1) empty indicating there is guaranteed to be no relationship between the abstract stack locations, (2) D indicating there is a *definite* relationship, or (3) P indicating that there is a *possible* relationship. As there is no approximation due to control flow in our example, we find that the points-to matrix in Figure 2(d) contains only empty and D entries.

[2] In this paper we concentrate on stack-allocated pointers, dynamically-allocated pointers will be handled using a variation of *path matrix analysis*[27, 29, 26].

Given the points-to matrix information at each program point, it becomes quite straightforward to define exactly which stack locations are written and read by each statement. Figure 2(e) summarizes the abstract stack locations read and written for our simple program, and Figure 2(f) gives the dependencies that are imposed by the reads and writes for each statement. Note that in the statement `temp_1 = *pp`, both the abstract location `pp` and the abstract location `p` are read.

It should be noted that our small example is intended as an introduction and it demonstrates only a small part of the points-to analysis methods. In fact, the complete interprocedural analysis properly handles both aliasing due to procedure calls with pointer arguments, function pointers and recursion [20]. The principal idea for procedures is that the correct number of abstract location names must be generated for each parameter that has a pointer type, and that the relationships between these generated names must safely approximate the actual relationships that hold at a call site. Thus, the generated abstract names are used to safely approximate all non-visible locations that may be affected by executing the procedure.[3] The complete analysis has been implemented for both scalars and structured data, and it is operational at the SIMPLE level in the McCAT C compiler.

For the purposes of this paper, the important point is that the location-based points-to analysis provides exactly the information required to define accurate read and write sets for each statement. By ensuring that every important location is safely represented in the points-to matrices, we can use the read and write sets as the basis for further analysis. Table 1 summarizes the read and write sets for each basic statement type in SIMPLE. Given a statement s, and a points-to matrix M, the table shows how to compute: (1) $W_p(s,M)$ - the set of abstract locations possibly written by s, (2) $W_d(s,M)$ - the set of abstract locations definitely written by s, and (3) $R(s,M)$ - the set of abstract locations read by s. Each abstract location corresponds to: (1) a scalar variable, (2) an array variable, (3) a structured variable, (4) a pointer variable, or (5) an anonymous variable name that corresponds to variables that are not visible in a procedure, but are reachable by dereferencing a variable that is visible. Note that in this table $P_d(x,M)$ denotes the set of abstract locations definitely pointed to by abstract location x (all abstract locations y such that $M[x,y] = D$), $P_p(x,M)$ denotes the set of abstract locations possibly pointed to by abstract location x (all abstract locations y such that $M[x,y] = P$), and $P(x,M)$ denotes $P_d(x,M) \cup P_p(x,M)$. Furthermore, we can define $W(s,M)$ to be $W_p(s,M) \cup W_d(s,M)$.

3 Constructing the α-TREE Representation

In the previous section we introduced the points-to matrix analysis that is performed on the SIMPLE intermediate representation, and we demonstrated how the points-to matrix information can be used to accurately

[3] This appears to be a generalization of the `nonvisible` name used in [33, 32].

s	$W_p(s,M)$	$W_d(s,M)$	$R(s,M)$
`x = y`	ϕ	{x}	{y}
`x =` *unop* `y`	ϕ	{x}	{y}
`x = y` *binop* `z`	ϕ	{x}	{y,z}
`*x = y`	$P_p(x,M)$	$P_d(x,M)$	{x,y}
`*x =` *unop* `y`	$P_p(x,M)$	$P_d(x,M)$	{x,y}
`*x = y` *binop* `z`	$P_p(x,M)$	$P_d(x,M)$	{x,y,z}
`x = &y`	ϕ	{x}	ϕ
`x = *y`	ϕ	{x}	{y} $\cup P(y,M)$

Table 1: Read and Write Sets for statement s relative to a points-to matrix M

calculate read and write sets for the basic statements in SIMPLE. In this section we discuss the various analyses and transformations that are required to transform SIMPLE trees to our new α-TREE representation.

3.1 An example program

Figure 3(a) provides a C program that demonstrates conditionals, loops, breaks and pointers. Figure 3(b) shows the resulting program after simplification and points-to analysis. Note that we have indicated the results of the points-to analysis at three important program points. At program point `A`, the points-to analysis indicates that abstract location `q` definitely points to `x`, and `p` definitely points to `y`. Similarly, at program point `B`, just before the `break` from the loop, the analysis reveals that both `q` and `p` definitely point to `y`. Just after the loop, at program point `C`, the analysis shows that `b` definitely points to `y`, while `q` points to either `x` or `y`.

Figure 3(c) shows how the results of points-to analysis can be used to replace pointer accesses with the appropriate scalar. Whenever a pointer variable *definitely* points to an abstract stack location corresponding to a variable, we can replace the pointer reference with the appropriate scalar. For example, the statement just before program point `A` we can replace `*p` with `y` and `*q` with `x`. Figure 3(c) gives the program after pointer-replacement has been applied to the entire program. In order to simplify further discussion, we have also applied copy-propagation and dead-code removal to give a smaller and cleaner example.

3.2 Constructing α-TREEs

The SIMPLE intermediate representation provides a natural way of representing nested regions of control structures. In SIMPLE the following basic constructs introduce a new region: **`if-else`**, **`switch-case`**, **`for`**, **`while`**, and **`do-while`**. In most cases control flows into the entry of the control region and out of the exit in the control region. In the case of **`break`** and **`continue`** statements, control goes to the exit or entry of the closest enclosing loop.

```
main()
{ int u, v, x, y;
  int *p,*q ;

  x = read_int() ;
  y = read_int() ;
  p = &y ;
  q = &x ;
  u = *p + *q ;

  while (u < 100)
    { v = read_int() ;
      if (u == v)
        { q = p ;
          p = &y ;
          break ;
        }
      u++ ;
      (*p)++ ;
    }

  if (v == 10)
    *q = *q + 1 ;
  else
    *p = *p + 1 ;

  return(*p + *q) ;
}
```

(a) original program

```
int main()
{ int u,v x,y;
  int  * p;
  int  * q;
  extern int read_int();

  int temp_0, temp_1, temp_2, temp_3;
  int temp_4, temp_5, temp_6, temp_7;

  x = read_int();
  y = read_int();
  p = (& y);
  q = (& x);
  temp_0 = (* p);
  temp_1 = (* q);
  u = (temp_0 + temp_1);

  /*-- POINT A:  -------*/
  /*-- (p->y), (q->x) --*/
  while ( (u <= 99) )
    { v = read_int();
      if ((u == v))
        { q = p;
          p = (& y);
          /*-- POINT B: --------*/
          /*-- (q->y), (p->y) --*/
          break;
        }
      u = (u + 1);
      temp_2 = (* p);
      (* p) = (temp_2 + 1);
    }
  /*-- POINT C: ------------------*/
  /*-- (p->y), (q->x)?, (q->y)? --*/

  if ((v == 10))
    { temp_3 = (* q);
      (* q) = (temp_3 + 1);
    }
  else
    { temp_4 = (* p);
      (* p) = (temp_4 + 1);
    }

  temp_6 = (* p);
  temp_7 = (* q);
  temp_5 = (temp_6 + temp_7);
  return (temp_5);
}
```

(b) after simplification and stack-matrix analysis

```
int main()
{ int u,v x,y;
  int  * q;
  extern int read_int();

  int temp_3, temp_5, temp_7;;

  x = read_int();
  y = read_int();
  q = (& x);
  u = (y + x);

  /*-- POINT A:  -------*/
  /*--  (q->x) --*/
  while ( (u <= 99) )
    { v = read_int();
      if ((u == v))
        { q = (& y);
          /*-- POINT B: --------*/
          /*-- (q->y) --*/
          break;
        }
      u = (u + 1);
      y = (y + 1);
    }
  /*-- POINT C: ------------------*/
  /*-- (q->x)?, (q->y)? --*/

  if ((v == 10))
    { temp_3 = (* q);
      (* q) = (temp_3 + 1);
    }
  else
    y = (y + 1);

  temp_7 = (* q);
  temp_5 = (y + temp_7);
  return (temp_5);
}
```

(c) after pointer-elimination, copy propagation, and dead-code removal

Figure 3: An example program with simplification and pointer-elimination

The central idea behind the α-TREE representation is that we wish to provide interception points for all dependences (flow, output, and anti) for locations that are read and/or written in the control region. This is accomplished by placing α-lists at the entry and exit of control regions. An α-list is simply a list of α-statements where each α-statement is a pseudo-assignment (identity assignment) of the form `l=`α`(l)`, where `l` is the name of an abstract location. We call the process of transforming a SIMPLE AST to an α-TREE , *α-decorating*. Figure 4 shows where the α-statements are placed relative to the original components for the **`if-else`** and **`while`** statements. Note that in both cases control must enter the region through the *inlist* (l_{in_1}=α(l_{in_1}); ...; l_{in_m}=α(l_{in_m});) and exit through the *outlist* (l_{out_1}=α(l_{out_1}); ...; l_{out_n}=α(l_{out_n});). In the case of the while loop at region k, any **`break`** statement in the body of the while loop would correspond to "**`goto exit_k`**" and any **`continue`** statement would correspond to "**`goto entry_k`**".

entry_k:
    $l_{in_1} = \alpha(l_{in_1})$;
    ...
    $l_{in_m} = \alpha(l_{in_m})$;
    if not < e >goto else_k;
    < s_then >
    goto exit_k;
else_k:
    < s_else >
exit_k:
    $l_{out_1} = \alpha(l_{out_1})$;
    ...
    $l_{out_n} = \alpha(l_{out_n})$;
	entry_k:
	    $l_{in_1} = \alpha(l_{in_1})$;
	    ...
	    $l_{in_m} = \alpha(l_{in_m})$;
	    if not < e >goto exit_k;
	    < s_body >
	    goto entry_k;
	exit_k:
	    $l_{out_1} = \alpha(l_{out_1})$;
	    ...
	    $l_{out_n} = \alpha(l_{out_n})$;

IF(inlist,e,s_then,s_else,outlist)	WHILE(inlist,e,s_body,outlist)

Figure 4: Alpha-decorated conditional and loop

By inserting the α-statements, we provide pseudo-assignments that force a *definite* use and a *definite* definition for each location.[4] Thus, just as the ϕ-functions in the SSA form provide a place for collecting dependences at the exit of control regions, the α-lists provide the appropriate places at both the entry and exit of control regions. It should be noted that these pseudo-assignments do not correspond to any instructions that must be generated, they are only a tool for further analysis. We only introduce new names when we explicitly wish to break output dependences.[5]

Given our SIMPLE tree representation, and the points-to analysis, the α-decorating procedure is straightforward and inexpensive to compute. Figure 5 summarizes the rules used to determine which locations should be captured by α-statements. The left column illustrates the rules for capturing an abstract location **`x`** on entry to a region, while the right column illustrates the rules for capturing **`x`** on exit of a region. The

[4] Note that we consider abstract location names and not variable names. This distinction is very important since it allows us to handle pointer variables very naturally.

[5] This is done using an extension of the renaming algorithm found in [46].

columns illustrate capture dues to *flow*, *anti*, and *output* dependences. In the figure, the square boxes correspond to conditional or loop statements which are denoted by S in the associated formula. Note that these formulae make use of *side-effect analysis* ($read(S)$ and $write(S)$), and *exposure analysis* ($will\_be\_read(S)$, $will\_be\_written(S)$, $was\_read(S)$, and $was\_written(S)$).

We demonstrate the simplicity and inexpensive nature of the capturing process by examining the capture of flow dependences exiting out of control regions (the upper right entry in Figure 5). In this case we need to know the set of abstract locations written by the region corresponding to statement S. As indicated in the figure, a location `x` should be captured if it is in the set $write(S) \cap will\_be\_read(S)$. We can compute the *write* set of any statement S by simply taking the union of the *write* sets of the components of S. For example, $write(IF(e, s\_then, s\_else)) = write(s\_then) \cup write(s\_else)$. The read and write sets can clearly be collected by one pass over the SIMPLE tree. The $will\_be\_read(S)$ set is the set of abstract locations that may be read (before redefinition) between the exit of S and the exit of the procedure. This corresponds to the normal notion of live variable analysis and can also be computed in $O(n)$ time (this algorithm is a variation of the structured flow analysis algorithm found in [7], and is given in full in Appendix A).

All the other sets required for capturing can also be computed in a similar manner. For these one calculates $will\_be\_written(S)$ - locations after the exit from S, $was\_read(S)$ - locations read before entry into S, and $was\_written(S)$ - locations written before entry into S.

We use the capturing process to transform a SIMPLE tree to an α-TREE . For each conditional and loop statement we store locations captured on entry in a list of α-statements, *inlist*, and locations captured on exit in *outlist*. In each case we use the rules as given in Figure 5 to determine which locations should be captured. The only exception is that a conditional that contains a break must inherit its *inlist* and *outlist* from the *inlist* of the closest enclosing loop. Figure 6 gives the α-TREE version of our example program. In the case of the `while` loop in this example, we see that the locations read are $\{u, v, y\}$, and the locations written are $\{u, v, q, y\}$. Using these read and write sets, plus the exposure analysis, we find that locations $\{u, y, q\}$ must be captured on entry by the inlist, and $\{y, q\}$ must be captured on exit by the outlist. Note that since `v` is only used inside the loop, it does not occur in either the inlist or outlist, and since `u` is not used after the loop, it does not appear in the outlist.

3.3 Computing α-Reaching Definitions

Once the α-TREE representation has been created, we have the complete infra-structure required to support efficient analyses (including dependence analysis). We illustrate this by presenting the solution for α-reaching-definitions.

Figure 7 gives the complete structured analysis for α-reaching-definitions. The major difference between this analysis and the ordinary reaching-definitions analysis is that we make use of the fact that all definitions

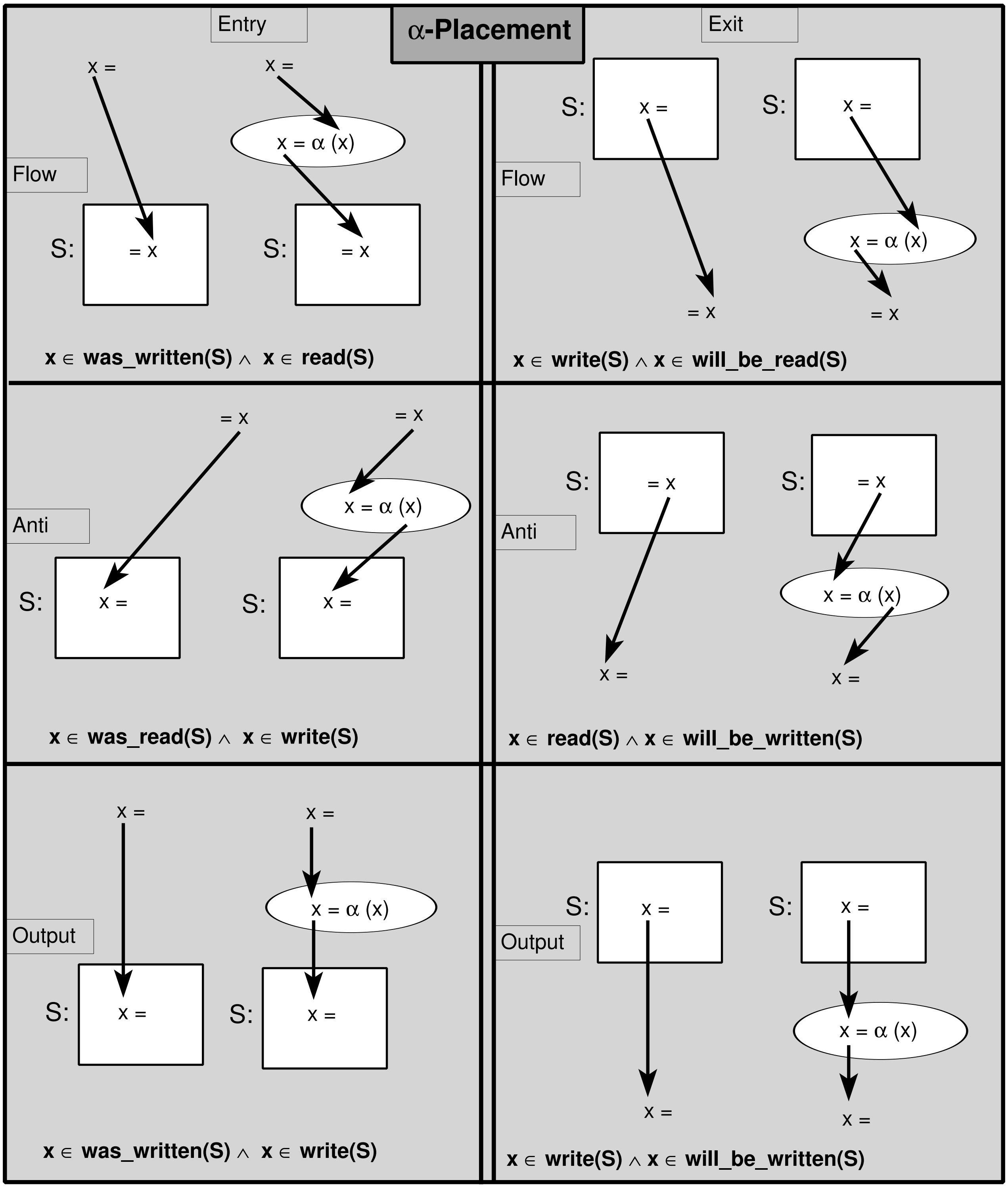


Figure 5: Capturing dependences with α-nodes

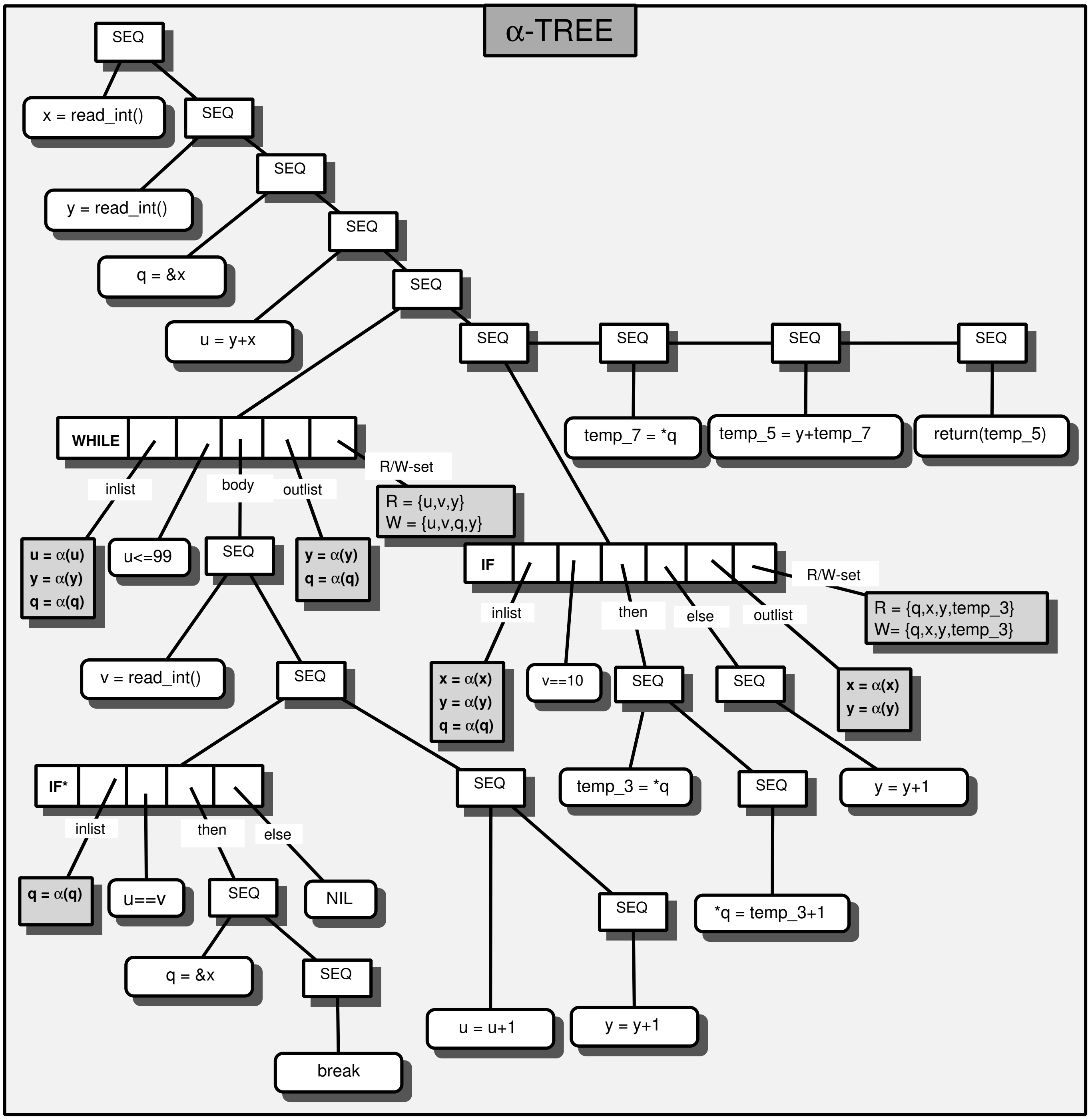


Figure 6: The α-TREE with R/W sets and α-lists

```
/* Given s - a statement,   in - the α-reaching_def set at the input of s,   and loop - the
   closest enclosing loop, compute the α-reaching_def for the components of s, and return the
   α-reaching_defs at the output of s. Each α-reaching_def is a pair <stmt#,l> where stmt# is a
   statement id, and l is a location being defined by that statement.
   As a side-effect, definitions reaching the input of statement s will be stored in s.in_rdefs, and definitions
   reaching break statements will be stored in s.break_rdefs where s in the closest enclosing loop for the break.
   For the IF and WHILE constructs, assume that the α-lists have already been inserted, and store
   the definitions reaching the input of the inlist (s.inlist_rdefs) and outlist (s.outlist_rdefs).
   Note: ⊥⋈⊥ →⊥ , ⊥⋈s →s, s ⋈⊥ →s, and s1 ⋈s2 →(s1 ∪ s2)  */

α-ReachDef(s,in,loop) =
{ case s of
    ASSIGN (lhs,rhs) ⇒
      { s.in_rdefs := in;
        kill_defs := { the set of all definitions <s#,l>∈ in, such that l ∈ W_d(s,s.SM) };
        gen_defs := { the set of all pairs <s.stmtid,l> such that l ∈ W(s,s.SM) };
        return(gen_defs ∪ (in - kill_defs));
      }
    SEQ (s1,s2) ⇒
      { s1.in_rdefs := in;
        s2.in_rdefs := α-ReachDef(s1,in,loop);
        if (s2.in_rdefs = ⊥ ) then return(⊥ ) else return(α-ReachDef(s2,s2.in_rdef,loop));
      }
    IF (inlist,e,s_then,s_else,outlist) ⇒
      { s.inlist_rdefs := { the set of all definitions <s#,l>∈ in, such that the statement l=α(l) ∈ inlist };
        in_defs := defs(inlist);
        s.outlist_rdefs := α-ReachDef(s_then,in_defs,loop) ⋈α-ReachDef(s_else,in_defs,loop);
        if (s.outlist_rdefs = ⊥ ) then
          return(⊥ );
        else
        { out_defs := defs(outlist);
          skip_defs := { the set of all definitions <s#,l>∈ in, such that the statement l=α(l) ∉ outlist };
          return(out_defs ∪ skip_defs);
        }
      }
    WHILE (inlist,e,s_body,outlist) ⇒
      { s.inlist_rdefs := { the set of all definitions <s#,l>∈ in, such that the statement l=α(l) ∈ inlist };
        in_defs := defs(inlist);
        s.inlist_rdefs := α-ReachDef(s_body,in_defs,s) ⋈s.inlist_rdefs;
        s.outlist_rdefs := in_defs ∪ s.break_rdefs;
        out_defs := defs(outlist);
        skip_defs := { the set of all definitions <s#,l>∈ in, such that the statement l=α(l) ∉ outlist };
        return(out_defs ∪ skip_defs);
      }
    CONTINUE ⇒
      { loop.inlist_rdefs := loop.inlist_rdefs ∪ in;   return(⊥ ); }
    BREAK ⇒
      { loop.break_rdefs := loop.break_rdefs ∪ in;   return(⊥ ); }
    NULLSTMT ⇒  return(in);
}
```

Figure 7: Computing the *alpha-reaching_def* sets for each statement

entering control regions must have been captured by an α-statement. This means that we propagate only the definitions generated by the α-statements of the inlist inside each control region. For example, consider the rule for IF(inlist,e,s_then,s_else,outlist) in Figure 7. The first action in this rule specifies that we should store in s.inlist_rdefs the set of all reaching definitions for locations that will be captured by α-statements in the inlist. These reaching definitions are used at a later stage to insert flow- and anti-dependences between definitions outside of the region and the appropriate α-statement capturing the dependence. The second statement, in_defs := defs(inlist);, specifies that we should create a set of α-reaching-def pairs, one pair for each α-statement in the inlist. A statement of the form *s_i: l=α(l)* corresponds to the pair < *s_i, l* >. The third statement uses in_defs as input and recursively applies the α-ReachDef function to the components s_then and s_else, and stores the merge ($\bowtie$) of the two output sets in s.outlist_rdefs. This set will be used to insert flow and output dependences between statements with the IF statement and the α-statements in the outlist. Finally, if the result of analyzing the components is not $\perp$ ($\perp$ indicates that a path was interrupted by a `break` or `continue` statement), then we return the reaching definitions generated by the α-statements of the outlist, along with any definitions that were not redefined (were skipped) by this IF statement. Note that any definitions generated inside the IF statement are captured by an α-statement in the outlist. The other rules in Figure 7 behave similarly.

ALPHA allows us to implement dataflow analyses in a simple and efficient manner. Our α-reaching_def analysis is efficient in time because it visits each node in the α-TREE only once (there is no need for iteration), and it is efficient in space because we propagate only the information needed for the inlist inside a region. In fact, we have developed a definition numbering system that allows us to propagate short bit-vectors inside each region. Another point to note is that the α-lists provide a natural barrier between regions, and provide a simple way to chop up a dataflow problem into independent pieces. Thus, this supports parallel flow analysis.

4 Implementation and Measurements

The SIMPLE representation had been completely implemented before developing the ALPHA representation. We have now also completed the first implementation of ALPHA and in this section we present our initial experience and measurements based on this implementation.

4.1 Implementing the SIMPLE-to-ALPHA translation

Each phase of the SIMPLE-to-ALPHA generation is implemented with a structured flow analysis (in the style of a practical abstract interpretation) where the abstraction is implemented with a bit-vector representation. Each bit in these vectors refers to either a SIMPLE statement number (definition or use point) or the index of an abstract location name. For the first stages of the transformation we must use full-length bit-vectors, while for the later stages we can take advantage of the α-statements to reduce the size of the bit vectors by including only the bits required for the abstract locations and statement numbers in each control region.

The first analysis is the read/write set estimation for each nested region. This is a straight-forward bottom-up traversal of the SIMPLE tree, where the read/write sets of a larger component is computed in terms of the read/write sets of its pieces.

The second major analysis is *exposure analysis* which consists of two types of analysis: 1) a backward analysis to estimate the *will-be-read* and *will-be-written* sets, and 2) a forward analysis to estimate the the *was-read* and *was-written* sets. Due to the structured nature of our approach then each analysis can be calculated by traversing each node in the SIMPLE tree once, and is thus linear.

Once the exposure analysis and read-write set for each constructs are calculated we use the method given in figure 5 to generate the inlist and outlist for each nested region. For each variable in the inlist and the outlist, an α-node is created. For structured programs it is trivial to identify program points to insert the inlist and the outlist. In the presence of `break`, `continue`, and `return` statements, the original SIMPLE syntactic information alone is not enough to identify points to insert the outlist (note that these "unstructured" constructs do not affect the insertion of the inlist). For such programs it is the case that `break/continue` (with the exception of `breaks` corresponding to the `switch-case` construct) occur inside a loop construct, and generally inside a conditional construct. We identify the conditional constructs that contain `break/continue/return` as a special conditional constructs. We label these with a special IF* notation, meaning that the `if` contains `break/continue/return` statements. We then use this special label to determine where to insert the α-statements corresponding to `break` and `continue`. The IF* conditional constructs have no outlist, but only an inlist. Any dependence crossing from inside these constructs are captured by outlist of the closest enclosing loop. Note that even with `break/continue` we still have the sequential compositionality of control flow for loops. We exploit this feature and perform structured flow analysis on SIMPLE intermediate representation for inserting the inlists and outlists.

Although we present our method for structured programs (our compiler uses a pre-processing step to structure programs with arbitrary gotos), the idea of ALPHA representation can be extended even to unstructured programs with arbitrary `goto` statements. For unstructured programs syntactic information alone is not sufficient to capture program points for inserting inlist and outlist. Here we first need to identify regions of identical control dependences and insert inlists and outlists whenever data dependences cross two regions. Since we have taken the approach of first structuring programs, we can take advantage of this structure, and thus implement straight-forward and efficient analyses for α-capturing.

Once we have inserted the inlist and the outlist we perform α-reaching definition to generate *def-use* chains. In our implementation we use the fact that no definition from outside a region can reach any statement inside the region, and also no definition from inside a region can reach any statements outside the region (these definitions are captured by the inlist and the outlist inserted at the region boundaries). This fact allows us to optimize the bit-vector length. For each region we use the appropriate bit-vector length to perform reaching definition. Once we have reaching definition information we build the def-use chains.

4.2 Experimental Measurements

We tested our implementation on 11 typical benchmark programs, 7 scientific programs and 4 non-scientific programs. The results of our measurements are summarized in Table 2. It should be noted that in Table 2 the number of statements (*Stmt*) refers to the number of statements at the SIMPLE level. This count includes only statements, and not variable and function declarations. The measurements IN and OUT give the total number of α-assignments introduced for all the inlists and outlists in the program. Note that in most benchmarks, the overhead due to the inlist and outlist is not very high as can be seen from the average number of in-out list with respect to both the number of statements (IO_{stmt}) and the number of regions (IO_{reg}).

The alias information abstracted using the points-to relation provide us with accurate results. We can estimate the effect of pointers by comparing the number of definite writes (W_d) with the number of possible writes (W_p). If these numbers are the same, then either the program has no pointers, or the points-to analysis gives exact results. If there is a large difference, then this reflects a program with many pointers where a pointer variable may point to more than one location. We note that 5 of the 11 benchmarks have aliasing due to pointers. In these 5 benchmarks we will have introduced α-statements to correctly capture dependences due to this aliasing behavior.

In `fft` and `deter`, arrays are used as pointers with a considerable amount of pointer arithmetic on array addresses. The points-to analysis makes conservative and a safe approximations when we deal with pointer arithmetic, and this leads to the introduction of more α-assignments in order to correctly capture all dependences. For programs that use array indexing rather than pointer arithmetic, we get a more accurate result for the points-to analysis, and thus fewer extra α-statements. This is can be seen in the benchmarks `clinpack`, `dhrys`, `knight`, `8queen`, and `tomcatv` programs. For these programs there is very little pointer arithmetic for manipulating arrays, and this leads to more definite writes. In `clinpack` there are procedures that pass addresses (of scalar variables) as parameters. Our points-to analysis computes very accurate alias information for these address parameters. If not for our powerful alias analysis we would get less accurate results and more α-statements in this case. The benchmark `flops` consists mostly of scalar operations. Alias analysis has no effect on this program. The benchmarks `nseive` and `3sort` treat arrays as pointers to heap-allocated objects. Most of the aliases in these two programs are due to these heap allocations. We notice that `dhrys` also allocates arrays from the heap. The reason that there are very few extra α-assignments in this program is because most aliases point to the heap, and we treat the whole heap as one abstract location. This situation will change when we incorporate a more detailed abstraction for the heap.

Generally the complexity of operations on bit-vectors is attributed to the length of the bit-vector. The

Name	*Proc*	*Reg*	*Stmt*	*IN*	*OUT*	W_d	W_p	*R*	IN_{max}	OUT_{max}	*IO*	IO_{stmt}	IO_{reg}	BV_{max}
clinpack	11	62	972	241	140	845	939	1601	12	11	381	0.39	6.14	139
tomcat	1	31	309	178	108	271	271	549	13	13	286	0.92	9.06	110
dhrys	13	16	194	25	13	140	150	257	5	7	38	0.30	2.37	42
deter	4	14	124	76	69	88	121	243	20	20	145	1.16	10.35	47
fft	2	19	265	354	280	212	851	1199	43	41	634	2.39	33.36	173
flop	1	13	720	115	140	686	686	1134	21	21	255	.35	19.61	75
livc	82	162	2326	617	227	1785	1944	2984	77	24	844	0.36	5.2	125
8queen	2	11	59	19	1	41	41	64	4	1	20	0.33	1.82	20
3sort	5	14	137	34	15	112	128	253	7	6	20	0.14	1.43	59
knight	4	15	99	71	9	69	71	112	29	3	80	0.81	5.33	64
nseive	2	18	187	93	52	143	165	211	16	14	145	0.77	8.05	70

Legend

Name	Name of the benchmark program
Proc	Total number of procedures in the program
Reg	Total number of regions in the program
Stmt	Total number of SIMPLE statements in the program
IN	Total number α-statements used in all in-lists
OUT	Total number α-statements used in all out-lists
W_d	Total number of definite writes in the program
W_p	Total number of possible writes in the program (W_d *is a subset of* W_p)
R	Total number of reads in the program
IN_{max}	Maximum size of the inlist for a region in the program
OUT_{max}	Maximum size of the outlist in the program
IO	In + Out Total number of α-statements introduced.
IO_{stmt}	Average number of IO per statement
IO_{reg}	Average number of IO per region
BV_{max}	Maximum size of a bitvector generated (for reaching definition)

Program Description

clinpack	LINPACK program
tomcatv	Tomcatv (mesh generation)
dhrys	Dhrystone
deter	Determinant using Laplace expansion
fft	Fast Fourier Transform
flop	Floating point rate estimator
livc	Livermore loops (C version)
8queen	The Eight Queens Problem
3sort	Bubble sort + Quick sort + Quick sort 3-median
knight	The Knight's Tour problem
nsieve	Sieve prime generator

Table 2: Experimental results

measurement BV_{max} gives the maximum size of the bit vector that we use for computing the α-reaching definitions. The small size of these bit vectors is very encouraging, in that it shows that we do get sparseness in the number of bits required per region. Comparing the BV_{max} with the number of statements per procedure (a lower-bound approximation of the number of bits required without our approach), we can get an estimate of the sparseness. In the case of `flop` and `tomcatv` this is easy to observe because there is only one procedure and we see that the maximum bit vector length is substantially less than the number of statements.

5 Applications

In this section we briefly discuss some of the most important applications that can be efficiently implemented using our ALPHA representation.

5.1 Applications of the SIMPLE Representation

The SIMPLE representation lends itself to the development of structured (compositional) analyses. In this paper we have summarized the more complex *points-to analysis*, and we have presented a very simple analysis for computing the *will_be_written* set. In our experience with the McCAT compiler, we have found SIMPLE to be a good setting for the development of a variety of analyses that are based on abstract interpretation techniques [40, 20]. In addition, we have used it for the development of an analyzer generator, McTAG [40].

5.2 Applications of the α-TREE Representation

As illustrated by the α-ReachDef algorithm presented in Figure 7, we can make use of the α-lists in the α-TREE representation to support structured analyses that are efficient in both time and space. The reaching definition and reaching use information can be used to efficiently compute dependence information in the α-TREEs , and thus provide the framework for implementing efficient algorithms that make use of the sparse dependence structure [25]. For example, we have studied how to adapt the techniques sparse constant propagation [42], and induction variable detection [45].

We also expect that we can improve some analyses that have previously been implemented on SIMPLE by exploiting both the sparse dependence information, and the region separation provided by the α-lists. Furthermore, based on our experience of implementing loop transformations such as unimodular transformation [39, 38], and collective loop optimization [22, 34], we believe that α-TREE representation provides a convenient and efficient platform for such loop optimizations.

6 Related Work

Our family of intermediate representations has been designed to benefit from a variety of techniques that apply to tree-based and/or graph-based representations.

Tree-based representations have traditionally been used to represent a program as a high-level abstract syntax tree that is used for semantic analysis, for source-to-source transformations, or as a source for lower-level intermediate code generation. Trees have also been used for program development tools [37], and in some dependence analyzer tools (parallelizing compilers) for performing loop transformations and other source-to-source transformations [8, 44]. With the advent of structured programming languages, automatic structuring techniques, and abstract interpretation analysis, it has become clear that many compile-time analyses naturally use a tree-like representation. Two compilers that follow this route include the MUG2 compiler [43], and the recent Miprac parallelizing compiler [23] that uses a low-level tree representation as the basis for a wide variety of interprocedural abstract interpretation analyses. Our SIMPLE and α-TREE representations allow us to fully exploit similar techniques. We believe that SIMPLE is particularly suited to the development of accurate alias analysis

In many compiler projects, the intermediate representation is based on some kind of graph. These include: *control-flow graphs* [7], *data dependence graphs* [31], *program dependence graphs* (PDG)[21] and *program dependence webs* [10], the *static single assignment form* [17, 18] and *sparse evaluation graphs* [14], *dataflow program graphs* [2, 41] and *dependence flow graphs* [35]. Our α-TREE representation has been designed to incorporate many of the interesting features of these other approaches.

The development of the α-TREE representation was influenced by the work on sparse representations based on single assignment principle [17, 14]. However, unlike the SSA form, we do not introduce unnecessary new names. We introduce names solely for the purpose of naming locations and removing (spurious) dependences. Furthermore, unlike the SSA form where one must process the ϕ-functions after analysis, one may generate code directly from the α-TREE representation by simply ignoring the α-statements. As we have demonstrated, the α-TREE representation provides an excellent framework to perform dataflow analyses and other optimizations efficiently at the desirable level of sparsity. We get sparseness and compactness both in terms of the evaluation process and as well as the size (i.e. bit-vector length) of the information that is being propagated to perform the analysis. Another major feature of our approach is that it has been designed, from the start, to handle programs with pointer data structures, and to represent dependence information both accurately and efficiently in the presence of pointer aliasing in such programs.

Another interesting representation is the dependence flow graph (DFG) approach [35, 30]. The DFG representation is designed to have a parallel execution model, and has an operational model which does not have sequential constraints besides those required by explicit dependence constraints. It also has the advantage of being a sparse graph which allow rapid traversal for efficient analysis and optimization. In

[25], we have demonstrated that, based on our ALPHA representation, it is straightforward to construct a structured graph representation similar to what has been described in [30].

7 Conclusions and Future Work

In this paper we have argued that it is important to develop dependence-based intermediate representations for languages like C, and that these representations must deal naturally with pointer data structures. We have introduced a new dependence-based representation that supports many of advantages of the SSA form, while also supporting pointers in a straight-forward manner.

We summarized SIMPLE, our base intermediate representation upon which we build the dependence-based representation, α-TREE . We pointed out that the abstract stack location approach to the points-to analysis, and the related read/write analysis for SIMPLE leads to efficient analyses that are required for building the dependence-based α-TREE representation. We have demonstrated all the steps necessary to build the α-TREE representation, and we have shown how the α-TREE dependence-based representation takes advantage of the structured nature of our intermediate representation to introduce the dependence capturing α-assignment nodes at both the entry and exit of each structured region. The relative sparseness of these α-assignment nodes then leads us to an efficient α-reaching definition analysis that is efficient in both time, and in space (as measured by the average length of bit-vector required).

We have completed the first implementation of the α-TREE representation. In our implementation we have exploited the sparseness characteristic of α-TREE to support analyses that are efficient in space and time. We have provided encouraging experimental measurements based on this implementation. These results show that accurate points-to information leads to a reasonable number of alpha nodes being introduced. Furthermore, the nested regions do give sparseness in the number of bits required for each region.

We intend to continue with this work by using the α-TREE representation for a variety of parallelizing transformations. In addition, we plan to extend the representation to include more precise alias information about heap-allocated structures.

References

[1] ACAPS technical memo, School of Computer Science, McGill University, Montréal, Québec.

[2] W. B. Ackerman. *Efficient Implementation of Applicative Languages.* PhD thesis, Massachusetts Institute of Technology, April 1984. Also published as Technical Report MIT/LCS/TR-323, MIT Laboratory for Computer Science.

[3] ACM SIGACT and SIGPLAN. *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 13–15, 1988.

[4] ACM SIGACT and SIGPLAN. *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, January 11–13, 1989.

[5] ACM SIGACT and SIGPLAN. *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 21–23, 1991.

[6] ACM SIGPLAN. *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, California, June 17–19, 1992. *SIGPLAN Notices,* 27(7), July 1992.

[7] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[8] J. R. Allen and K. Kennedy. PFC: A program to convert FORTRAN to parallel form. Technical Report MASC TR82-6, Rice University, Houston, TX, March 1982.

[9] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 13–15, 1988. ACM SIGACT and SIGPLAN.

[10] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, White Plains, New York, June 20–22, 1990. ACM SIGPLAN. *SIGPLAN Notices,* 25(6), June 1990.

[11] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages.* ACM SIGACT and SIGPLAN, January 1979.

[12] Jeffrey M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, September 1978.

[13] J. D. Choi, M. G. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[14] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 21–23, 1991. ACM SIGACT and SIGPLAN.

[15] Keith Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, New Orleans, Louisiana, January 13–16, 1985. ACM SIGACT and SIGPLAN.

[16] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–59, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[17] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single-assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[18] 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.

[19] Ron Cytron, Andy Lowry, and Kenneth Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, St. Petersburg Beach, Florida, January 13–15, 1986. ACM SIGACT and SIGPLAN.

[20] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, Montréal, Québec, September 1993.

[21] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[22] G. R. Gao, R. Olsen, V. Sarkar, and R Thekkath. Collective loop fusion for array contraction. In *5th Workshop on Languages and Compilers for Parallel Computing*. August 1992. Also to appear in LNCS.

[23] W. L. Harrison III and Z Ammarguellat. A program's eye view of miprac. In *5th Workshop on Languages and Compilers for Parallel Computing*. August 1992. Also to appear in LNCS.

[24] L. J. Hendren, C. Donawa, M. Emami, G. Gao, J. Justiani, and B Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *5th Workshop on Languages and Compilers for Parallel Computing*. August 1992. Also to appear in LNCS.

[25] L. J. Hendren, G. R. Gao, and V. C. Sreedhar. ALPHA: A dependence-based intermediate representation for an optimizing/parallelizing C compiler. ACAPS Technical Memo 49, School of Computer Science, McGill University, Montréal, Québec, November 1992.

[26] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 249–260, San Francisco, California, June 17–19, 1992. ACM SIGPLAN. *SIGPLAN Notices,* 27(7), July 1992.

[27] Laurie J. Hendren. *Parallelizing Programs with Recursive Data Structures.* PhD thesis, Cornell University, January 1990.

[28] Laurie J. Hendren, Guang R. Gao, Erik R. Altman, and Chandrika Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In U. Kastens and P. Pfahler, editors, *Proceedings of the International Conference on Compiler Construction*, number 641 in Lecture Notes in Computer Science, pages 176–191. Springer-Verlag, October 1992.

[29] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[30] Richard Johnson and Keshav Pingali. Dependence-based program ananlysis. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, 1993.

[31] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Analysis and transformation of programs for parallel computation. In *Proceedings of the Fourth International Computer Software and Application Conference*, October 1980.

[32] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Symposium on Programming Language Design and Implementation*, pages 235–248, 1992.

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

[34] R. Olsen. Collective analysis and transformation of loop clusters. Master's thesis, McGill University, May 1992.

[35] K. Pingal, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence Flow Graph: An algebriac approach to program dependencies. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 21–23, 1991. ACM SIGACT and SIGPLAN.

[36] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, California, January 13–15, 1988. ACM SIGACT and SIGPLAN.

[37] David S. Rosenblum. A Methodology for the Design of Ada Transformation Tools in a DIANA Environment. *IEEE Software*, pages 24–33, 1985.

[38] N. Shiri and G. R Gao. Unimodular transformations of monolithic array constructors. ACAPS Technical Memo 43, School of Computer Science, McGill University, Montréal, Québec, June 1992.

[39] Nematollaah Shiri-Varnaamkhaasti. A design and implementation of unimodular transformations of loops. Master's thesis, McGill University, Montréal, Québec, May 1992.

[40] Bhama Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill University, Montréal, Québec, September 1992.

[41] Kenneth R. Traub. A compiler for the MIT tagged-token dataflow architecture. Master's thesis, Massachusetts Institute of Technology, August 1986. Also published as Technical Report MIT/LCS/TR-370, MIT Laboratory for Computer Science.

[42] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[43] R. Wilhelm. Global flow analysis and optimization in the MUG2 compiler generating system. *Program Flow Analysis: Theory and Applications*, 1981.

[44] M. Wolfe. The Tiny loop restructing research tool. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, St. Charles, Illinois, August 12–16, 1991.

[45] M. Wolfe. Beyond induction variables. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, San Francisco, California, June 17–19, 1992. ACM SIGPLAN. *SIGPLAN Notices,* 27(7), July 1992.

[46] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.

A Structured Flow Analysis for the *will_be_read* Sets

The purpose of this appendix is to give the algorithms for computing the *will_be_written* sets for each statement in a SIMPLE tree. These can be computed in two passes over the tree. In the first pass, the algorithm presented in Figure 8 is used to collect the gen sets for each statement in the tree. The algorithm proceeds from the bottom leaf nodes upwards through the compositional nodes.

Given that the gen sets have been calculated and stored in the field s.gen, the algorithm presented in Figure 9 can be used to compute the *will_be_written* sets for each statement in a top down fashion. Note that no fixed-point iteration is required, and thus these sets can be computed by one pass down through the tree.

```
{ Given a statment s, compute the gen_set bottom-up.
  Store the set in s.gen.  }

GenSet(s) =
{ case s of
    ASSIGN (lhs,rhs) ⇒
      { s.gen := R(s,s.SM); }
    SEQ (s1,s2) ⇒
      { GenSet(s1);   GenSet(s2);
        s.gen := s1.gen ∪ s2.gen ;
      }
    IF (e,s_then,s_else) ⇒
      { GenSet(s_then);   GenSet(s_else);
        s.gen := s_then.gen ∪ s_else.gen ∪ R(e,e.SM);
      }
    IF* (e,s_then,s_else) ⇒
      { GenSet(s_then);   GenSet(s_else);
        s.gen := s_then.gen ∪ s_else.gen ∪ R(e,e.SM);
      }
    WHILE (e,s_body) ⇒
      { GenSet(s_body);
        s.gen := s_body.gen ∪ R(e.e.SM);
      }
    CONTINUE ⇒  { s.gen := φ; }
    BREAK ⇒  { s.gen := φ; }
    NULLSTMT ⇒  { s.gen := φ; }
}
```

Figure 8: Computing the gen set for the *will_be_written* problem

```
{ Given a statement s, the will_be_written set at the output of s (out), and
  the closest enclosing loop (loop), return the will_be_written at the
  input of s. As a side-effect, store the will_be_written set for the exit
  of each IF, and the break-exit and normal-exit of each WHILE.
  Note that the gen set for each node should have been already calculated. }
  /* Merge-rules: x ⋈ x = x ; x ⋈ y = x ∪ y;
       x ⋈ ⊥ = ⊥ ⋈ x = ⊥; ∅ ⋈ x = x */
WIW(s,out,loop) =
{ case s of
    ASSIGN (lhs,rhs) ⇒
      return(s.gen ⋈ out );
    SEQ (s1,s2) ⇒
      { in_2 = WIW(s2,out,loop);
        return(WIW(s1,in_2,loop));
      }
    IF (e,s_then,s_else) ⇒
      { s.exit_wiw = out;
        in_then = WIW(s_then,[out.N;∅],loop);
        in_else = WIW(s_else,[out.N;∅],loop);
          return(in.then ∪in.else ∪ R(e,e.SM)) ;
      }
    IF* (e,s_then,s_else) ⇒
      { s.exit_wiw = ∅;
        in_then = WIW(s_then,[out.N;∅],loop);
        in_else = WIW(s_else,[out.N;∅],loop);
          return(R(e,e.SM)) ;
      }
    WHILE (e,s_body) ⇒
      { s.break_wiw = out;   s.exit_wiw = out ∪ s.gen;
        in_body = WIW(s_body,s.exit_wiw,s);
        return(in_body ∪ R(e,e.SM));
      }
    CONTINUE ⇒
      { s.exit = loop.exit_wiw ;
        return(loop.exit_wiw);
      }
    BREAK ⇒
      { s.exit = loop.break_wiw ;
        return(loop.break_wiw);
      }
    NULLSTMT ⇒  return(out);
}
```

Figure 9: Computing the will_be_written sets

Summary

The effective compilation of many important C programs, including some of the so-called "grand challenge" problems, requires sophisticated compile-time analyses and optimizing/parallelizing transformations. The choice of intermediate representation greatly impacts the development of analyses and transformations. For example, the recent development of representations like the SSA form [17], and *sparse data flow evaluation graphs* [14], has led to the development of efficient analyses that use these forms [36, 19, 9, 42]. The main motivation of our work was to investigate and develop an intermediate representation that would allow us to support a wide variety of analyses and optimizing/parallelizing transformations in a straightforward and efficient manner. On one hand, one requires a representation that lends itself to the development and application of accurate alias and dependence analysis. On the other hand, one would like dependence-based representations that support a wide-variety of efficient (sparse) analyses including those that have been successfully developed for the SSA form and other sparse representations.

An important facet of our work is that we wanted intermediate representations that can handle imperative programs including programs that use pointer data structures. Thus, we require both an intermediate representation that supports alias analysis in the presence of pointers, and a dependence-based intermediate representation that can represent dependences due to pointer aliasing in a natural fashion.

The development of our representations has been motivated by our experiences in building the McCAT C compiler, an optimizing/parallelizing compiler targeted at high-performance architectures that support fine-grain parallelism and/or multi-threading [24]. Our compiler makes use of three structured intermediate representations FIRST, SIMPLE, and LAST [24]. Each representation is a structured AST that fulfills a specific role in the compiler.

In our previous work, we have presented the design and implementation of SIMPLE and LAST [28]. In this paper we focus on the development of the ALPHA dependence-based representation. We start with the SIMPLE representation, and through various analyses and transformations we create the skeleton of the α-TREE , and then finally the α-TREE with dependences.

The main contributions of this paper are:

- We briefly summarize the foundations upon which we build: (1) the SIMPLE intermediate representation, and (2) the interprocedural *points-to analysis* that is used to compute accurate dependence information even in the presence of stack-based pointers. The SIMPLE representation and the points-to analysis makes use of the concept that every important location should have a name. This feature enables us to build the dependence-based representations in a straightforward manner.
- We introduce a new type of structured intermediate representation, α-TREEs and we demonstrate how this representation can be efficiently constructed from SIMPLE. The α-TREE representation differs from SIMPLE in that it includes α-statements that capture dependences entering and exiting structured control regions corresponding to conditionals and loops. This intermediate form is efficiently and correctly constructed even in the presence of pointers, `break` and `continue` statements in loops, and recursion. Furthermore, we show how the α-TREE representation lends itself to the development of efficient analyses. In particular, we show how reaching definitions can be efficiently implemented in both space and time. We also discuss other applications of α-TREEs for sparse analyses.
- We provide experimental data to illustrate various characteristics of the α-TREE representation.