1998

Type-Based Alias Analysis

Amer Diwan
Stanford University

Kathryn S. McKinley
University of Massachusetts - Amherst

J. Eliot B. Moss
University of Massachusetts - Amherst

Type-Based Alias Analysis*

Amer Diwan Kathryn S. McKinley J. Eliot B. Moss

Department of Computer Science
Stanford University, Stanford, CA 94305
(650) 723-4013

Department of Computer Science
University of Massachusetts, Amherst, MA 01003-4610

Abstract

This paper evaluates three alias analyses based on programming language types. The first analysis uses type compatibility to determine aliases. The second extends the first by using additional high-level information such as field names. The third extends the second with a flow-insensitive analysis. Although other researchers suggests using types to disambiguate memory references, none evaluates its effectiveness. We perform both static and dynamic evaluations of type-based alias analyses for Modula-3, a statically-typed type-safe language. The static analysis reveals that type compatibility alone yields a very imprecise alias analysis, but the other two analyses significantly improve alias precision. We use redundant load elimination (RLE) to demonstrate the effectiveness of the three alias algorithms in terms of the opportunities for optimization, the impact on simulated execution times, and to compute an upper bound on what a perfect alias analysis would yield. We show modest dynamic improvements for (RLE), and more surprisingly, that on average our alias analysis is within 2.5% of a perfect alias analysis with respect to RLE on 8 Modula-3 programs. These results illustrate that to explore thoroughly the effectiveness of alias analyses, researchers need static, dynamic, and upper-bound analysis. In addition, we show that for type-safe languages like Modula-3 and Java, a fast and simple alias analysis may be sufficient for many applications.

1 Introduction

To exploit memory systems, multiple functional units, and the multi-issue capabilities of modern uniprocessors, compilers must reorder instructions. For programs that use pointers, the compiler's alias analysis dramatically affects its ability to reorder instructions, and ultimately performance. Alias analysis disambiguates memory references, enabling the compiler to reorder statements that do pointer accesses.

Despite its importance, few commercial or research compilers implement non-trivial alias analysis. Three reasons alias analysis is not implemented are: (1) Many alias analyses are prohibitively slow and thus impractical for production use. (2) The alias analyses in the literature require the entire program (or some representation of it), which inhibits separate compilation and compiling libraries. (3) Most alias analyses have been evaluated only statically, and thus we do not know the effectiveness of these algorithms with respect to the optimizations that use them. To address these concerns, this paper explores using fast alias analyses that rely on programming language types. While prior work [1, 6] mentions using type compatibility for alias analysis, none evaluates the idea or presents the details of an algorithm.

This paper describes and evaluates three fast alias analyses based on programming language types. The first analysis (*TypeDecl*) uses type compatibility to determine aliases. The second (*FieldTypeDecl*) uses other high-level properties, such as field names to improve on the first. The third (*SM-FieldTypeRefs*) improves the second by incorporating a flow-insensitive pass to include the effects of variable assignments and references. This pass is similar to Steensgaard's algorithm [32].

We statically evaluate our alias algorithms using the number of alias pairs (the traditional method). We also evaluate TBAA based on its static and dynamic effects on an optimization. In addition, we evaluate TBAA with respect to an upper bound on the same optimization. Each of the evaluation metrics reveals different strengths and weaknesses in our alias algorithms, and we believe this range of metrics, and especially upper-bound analysis, is necessary to understand the effectiveness of any alias analysis.

Our static evaluation reveals that the simplest type-based alias analysis is very imprecise, but that for our Modula-3 benchmarks, the other two alias analyses significantly reduce the number of intraprocedural aliases of a reference to on average 3.4 references (ranging from .3 to 20.8). We find that TBAA is much less effective for interprocedural aliases.

We also evaluate TBAA by measuring the static and simulated run-time impact on an intraprocedural optimization that depends on alias analysis: *redundant load elimination* (RLE). RLE combines loop invariant code motion and common subexpression elimination of memory references. TBAA and RLE combine to improve simulated program performance modestly, by an average of 4%, and up to 8% on a DEC Alpha 3000-500 [12] for 8 Modula-3 benchmarks.

*The authors can be reached electronically via Internet addresses diwan@cs.stanford.edu, {mckinley,moss}@cs.umass.edu. This work was supported by the National Science Foundation under grants CCR-9211272 and CCR-9525767 and by gifts from Sun Microsystems Laboratories, Inc., Hewlett-Packard, and Digital Equipment. Kathryn S. McKinley is supported by an NSF CAREER Award CCR-9624209. Amer Diwan was also supported by the Air Force Materiel Command and ARPA award number: F30602-95-C-0098.

In the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1998, Montreal, Quebec, Canada, pp. 106–117.

We also compare TBAA to an upper bound that represents the best any alias analysis algorithm could hope to do for RLE. This comparison shows that a perfect alias analysis could at most eliminate an average of 2.5% more heap loads. In addition, we modify TBAA for incomplete programs and demonstrate, using RLE, that it performs as well as it does on complete programs. These results and TBAA's fast time complexity suggest that TBAA is a practical and promising analysis for scalar optimization of type-safe programs.

The remainder of this paper is organized as follows. Section 2 describes our type-based alias analysis algorithms. Section 3 presents our evaluation methodology, and uses it to evaluate TBAA. Section 4 extends and evaluates TBAA for incomplete programs. Section 5 discusses related work in alias analysis. Section 6 concludes.

2 Type-Based Alias Analysis

This section describes type-based alias analyses (TBAA) in which the compiler has access to the entire program except for the standard libraries. TBAA assumes a type-safe programming language such as Modula-3 [25] or Java [33] that does not support arbitrary pointer type casting (this *feature* is supported in C and C++). We begin with our terminology, and then discuss using type declarations, object field and array access semantics, and modifications to the set of possible types via variable assignments to disambiguate memory accesses.

2.1 Memory Reference Basics

Table 1 lists the three kinds of memory references in Modula-3 programs, their names, and a short description.[1]

Table 1: Kinds of Memory References

Notation	Name	Description
`p.f`	Qualify	Access field `f` of object `p`
`p^`	Dereference	Dereference pointer `p`
`p[i]`	Subscript	Array `p` with subscript `i`

We call a non-empty string of memory references, for example `a^.b[i].c`, an *access path* ($\mathcal{AP}$) [22]. Without loss of generality, we assume that distinct object fields have different names. We also define:

Type (`p`):	The static type of $\mathcal{AP}$ `p`.
Subtypes (`T`):	The set of subtypes of type `T`, which includes `T`.

In Modula-3 and other type-safe languages, an object of type `T` can legally access objects of type *Subtypes* (`T`). Each of our alias analyses refines the type of objects to which an $\mathcal{AP}$ (memory reference) may refer. If two $\mathcal{AP}$s may have the same type, then the analyses determines they may access the same location.

[1] These types of memory references are, of course, not unique to Modula-3.

```
TYPE
   T = OBJECT f, g: T; END;
   S1 = T OBJECT ... END;
   S2 = T OBJECT ... END;
   S3 = T OBJECT ... END;
VAR
   t: T;
   s: S1;
   u: S2;
```

Figure 1: Type Hierarchy Example

2.2 TBAA Using Type Declarations

To use type declarations to disambiguate memory references, we simply examine the declared type of an access path $\mathcal{AP}$, and then assume the $\mathcal{AP}$ may reference any object with the same declared type or subtype. We call this version of TBAA, *TypeDecl*. More formally, given two $\mathcal{AP}$s `p` and `q`, *TypeDecl*(`p`, `q`) determines they may be aliases if and only if:

$$\textit{Subtypes}\ (\textit{Type}\ (\texttt{p})) \cap \textit{Subtypes}\ (\textit{Type}\ (\texttt{q})) \neq \emptyset.$$

Consider the example in Figure 1. Since `S1` is a subtype of `T`, objects of type `T` can reference objects of type `S1`. Thus,

$$\textit{Subtypes}\ (\textit{Type}\ (\texttt{t})) \cap \textit{Subtypes}\ (\textit{Type}\ (\texttt{s})) \neq \emptyset$$
$$\textit{Subtypes}\ (\textit{Type}\ (\texttt{t})) \cap \textit{Subtypes}\ (\textit{Type}\ (\texttt{u})) \neq \emptyset$$
$$\textit{Subtypes}\ (\textit{Type}\ (\texttt{s})) \cap \textit{Subtypes}\ (\textit{Type}\ (\texttt{u})) = \emptyset$$

In other words, `t` and `s` may reference the same location, and `t` and `u` may reference the same location, but `s` and `u` may not reference the same location since they have different types. Note that *TypeDecl* is not transitive.

Table 2: *FieldTypeDecl* ($\mathcal{AP}$ 1, $\mathcal{AP}$ 2) Algorithm

Case	$\mathcal{AP}$ 1	$\mathcal{AP}$ 2	*FieldTypeDecl* ($\mathcal{AP}$ 1, $\mathcal{AP}$ 2)
1	`p`	`p`	true
2	`p.f`	`q.g`	(`f` = `g`) ∧ *FieldTypeDecl* (`p`, `q`)
3	`p.f`	`q^`	AddressTaken (`p.f`) ∧ *TypeDecl* (`p.f`, `q^`)
4	`p^`	`q[i]`	AddressTaken(`q[i]`) ∧ *TypeDecl* (`p^`, `q[i]`)
5	`p.f`	`q[i]`	false
6	`p[i]`	`q[j]`	*FieldTypeDecl* (`p`, `q`)
7	`p`	`q`	*TypeDecl* (`p`, `q`)

2.3 Using Field Access Types

We next improve the precision of *TypeDecl* using the type declarations of fields and other high level information in the program. We call this version of type-based alias analysis *FieldTypeDecl*. It distinguishes accesses such as `t.f` and `t.g`, `f` $\neq$ `g`, that *TypeDecl* misses. The *FieldTypeDecl* algorithm appears in Table 2. Given $\mathcal{AP}$`1` and $\mathcal{AP}$`2`, it returns true if $\mathcal{AP}$`1` and $\mathcal{AP}$`2` may be aliases. It uses *AddressTaken*, which returns true if the program ever takes the address of its argument. For example, *AddressTaken* (`p.f`) is true if the program takes the address of field `f` of an object in the set *TypeDecl* (`p`). *AddressTaken* (`q[i]`) returns true if the program takes the address of some element of an array of `q`'s type. In Modula-3, programs may take the addresses of memory locations in only two ways: via the pass-by-reference parameter passing mechanism, and via the `WITH` statement, which creates a temporary name for an expression. For simplicity we assume that aggregate accesses, such as assignments between two records, have been broken down into accesses of each component.

The seven cases in Table 2 determine the following.

- **1**: Identical $\mathcal{AP}$s always alias each other.
- **2**: Two qualified expressions may be aliases if they access the same field in potentially the same object.
- **3-4**: A pointer dereference may reference the same location as a qualified or subscripted expression only if their types are compatible and the program may take the address of the qualified or subscripted expression.
- **5**: In Modula-3, a subscripted expression cannot alias a qualified expression.
- **6**: Two subscripted expressions are aliases if they may subscript the same array. *FieldTypeDecl* ignores the actual subscripts.
- **7**: For all other cases of $\mathcal{AP}$s, including two pointer dereferences, *FieldTypeDecl* uses *TypeDecl* to determine aliases.

Java programs would have similar rules. For C++ programs, the rules must be more conservative to handle arbitrary pointer casts and pointer arithmetic.

2.4 Using Assignment

TypeDecl is conservative in the sense that it assumes that the program uses types in their full generality. For instance, programs often use list packages that support linking objects of different types to link objects of only one type. We thus improve on *TypeDecl* by examining the effects of explicit and implicit assignments to determine more accurately the types of objects an $\mathcal{AP}$ may reference in a flow-insensitive manner. We call this algorithm *SMTypeRefs* (*Selectively Merge Type References*). Unlike *TypeDecl*, which always merges the declared type of an $\mathcal{AP}$ with all of its subtypes, *SMTypeRefs* only merges a type with a subtype when a statement assigns some reference of subtype `S` to a reference of type `T`. As an example, consider applying *TypeDecl* to the following program given the type hierarchy in Figure 1:

```
VAR
  t: T := NEW (T);
  s: S1 := NEW (S1);
```

Since *TypeDecl* only considers declared types, it assumes that `t` and `s` may reference the same location because it is semantically correct for objects of type `T` to reference objects of type `S1`. By inspecting the code however, it is obvious that `t` and `s` never reference the same location since there is no explicit or implicit assignment between the two. *SMTypeRefs* proves independence in this situation as follows: if the program never assigns an object of type `S1` to a reference of type `T` (directly or indirectly), then `t` and `s` cannot possibly be aliases. Notice that if there is any such assignment, *SMTypeRefs* assumes that $\mathcal{AP}$s of type `T` may be aliased to $\mathcal{AP}$s of type `S1`. We call these assignments *merges*.

```
(* Step 1: put each type in its own set *)
for all pointer types T do
  Group := Group + {{T}}

(* Step 2: merge sets because of assignments *)
for all implicit and explicit pointer assignments, a:=b, do
  Ta := Type (a);    Tb := Type (b);
  if Ta ≠ Tb then
    let Ga, Gb ∈ Group, such that Ta ∈ Ga, Tb ∈ Gb
      Group := Group - {Ga} - {Gb} + {Ga ∪ Gb}

(* Step 3: Construct TypeRefsTable *)
for all types t do
  let g ∈ Group, t ∈ g
    TypeRefsTable (t) = g ∩ Subtypes (t)
```

Figure 2: Selective Type Merging

Figure 2 presents the algorithm to selectively merge types.[2] This algorithm produces a *TypeRefsTable* which takes a declared type `T` as an argument and returns all the types potentially referenced by an $\mathcal{AP}$ declared to be of type `T`. Given two $\mathcal{AP}$ `p` and `q`, *SMTypeRefs* (`p`,`q`) determines they may be aliases if and only if:

$$\textit{TypeRefsTable}\ (\textit{Type}\ (\texttt{p})) \cap \textit{TypeRefsTable}\ (\textit{Type}\ (\texttt{q})) \neq \emptyset$$

In Figure 2, each set $S = \{T_1, \ldots, T_k\}$ in `Group` represents an equivalence class of types such that an $\mathcal{AP}$ with a declared type $T \in S$ may reference any objects of type $T_i \in S$. For example, given the set $S = \{$`T1`,`T2`$\} \in$ `Group`, $\mathcal{AP}$s with declared type `T1` may reference any object of type `T1` or `T2`.

Step 1 initializes `Group`, such that each declared type is in an independent set and an $\mathcal{AP}$ declared with type `T` is thus assumed to reference only objects of type `T`. Step 2 examines all the assignment statements and merges the type sets if the types of the left and right hand sides are different.[3] Step 2 does not consider the order of the instructions and is therefore *flow insensitive*. Step 3 then filters out infeasible aliases from `Group`, creating *asymmetry* in the *SMTypeRefs* relationship.[4] For instance, an $\mathcal{AP}$ with declared type `T` in Figure 1 may reference objects of type `T` or type `S1`, but an $\mathcal{AP}$ declared as `S1` may not reference objects of type `T`. The final result of Step 3 is the *TypeRefsTable*.

Figure 3 uses the the type declarations in Figure 1 to illustrate how the selective merging algorithm works. The

[2] A more precise but slower formulation maintains a separate group for each type. In our experiments, the difference between the two variations was insignificant.

[3] Step 2 is similar to Steensgaard's algorithm [32].

[4] If we took Steensgaard's algorithm [32] and applied it to user defined types, it would not discover this asymmetry.

```
VAR
    s1: S1 := NEW (S1);
    s2: S2 := NEW (S2);
    s3: S3 := NEW (S3);
    t: T;
BEGIN
    t := s1; (* Statement 1 *)
    t := s2; (* Statement 2 *)
END;
```

Figure 3: Example to Illustrate *SMTypeRefs*

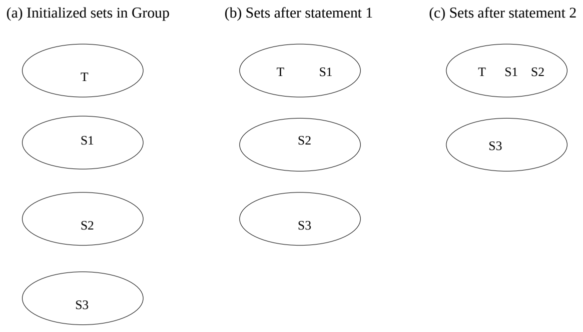
(a) Initialized sets in Group
(b) Sets after statement 1
(c) Sets after statement 2
T
S1
S2
S3
T S1
S2
S3
T S1 S2
S3

Figure 4: Selective Merging for Figure 3

VAR declarations declare and initialize variables to newly allocated objects of their declared types. Step 1 thus initializes each declared type in a set of its own, as shown in Figure 4(a) where each oval represents a set in Group. Figure 4(b) shows Group after Step 2 merges types T and S1, the types for the first assignment; and Figure 4(c) shows that the second assignment causes Step 2 to merge S2 with T and S1. S3 remains in a set by itself. Step 3 of the merge algorithm then creates asymmetry for the subtype declarations in the *TypeRefsTable*, as shown in Table 3. Notice *SMTypeRefs* determines $\mathcal{AP}$s declared to be of type T may not reference objects of type S3, but *TypeDecl* must assume they may.

Table 3: *TypeRefsTable* for Figure 3

Type	*TypeRefsTable* (Type)
T	T, S1, S2
S1	S1
S2	S2
S3	S3

We obtain the final version of our TBAA algorithm *SMFieldTypeRefs* (*Fields+Selectively Merge Type References*) by using *SMTypeRefs* for *TypeDecl* in the *FieldTypeDecl* algorithm in Table 2.

2.5 Complexity

The complexity of this type-based alias analysis (TBAA) is dominated by step 2 of *SMTypeRefs*. This step makes a single linear pass through the program and at each pointer assignment unions two sets of types. The complexity of TBAA is thus $O(n)$ bit-vector steps, where n is the number of instructions in the program. Each bit-vector step takes time proportional to the number of types in the program. The time to *use* the results of the TBAA may, of course, be more than linear time. For instance, computing all the *may-alias* pairs using TBAA (or any other *points-to* analysis) takes $O(\mathrm{e}^2)$ time, where e is the number of memory expressions in the program.

3 Evaluation

This section evaluates type-based alias analysis using static and dynamic metrics, and a *limit* analysis. We first review the strengths and weaknesses of static and dynamic metrics, and thus motivate our limit analysis.

Static Evaluation. The majority of previous work on alias analysis [2, 4, 6, 7, 9, 15, 20, 21, 22, 30, 32, 35] measures *static properties*, such as the sizes of the *may alias* and *points-to* sets. Static properties enable comparisons between the precision of two alias analyses using the size of their static points-to sets; the smaller the set the more precise the analysis. Static properties have, however, two main disadvantages. (1) Static properties cannot tell us if the analysis is effective with respect to its clients. For example, even if the alias sets are small, the analysis may not differentiate the pointers that will enable optimizations to improve performance or increase the effectiveness of other analyses. (2) Static properties do not enable comparisons between the *effectiveness* of two alias analyses with different strengths and weaknesses. For example, the size of the points-to sets of two analyses may be the same, but the analyses may disambiguate different pointers. A static analysis that compares the resulting number of optimization opportunities remedies some of this problem.

Dynamic Evaluation. A few researchers recently evaluated alias analyses by measuring the *execution-time improvement* due to an optimization that uses alias analysis [19, 36, 8, 17]. Using run-time improvements complements static metrics, since run-time improvements directly measure the impact of the alias analysis on its clients (usually compiler optimizations). However, one of their disadvantages is that the results are specific to the given program inputs.

Limit Evaluation. Both static and dynamic evaluation have an additional significant shortcoming: these properties do not tell us how much room for improvement there is in the alias analysis (except in the unusual case of an alias analysis that disambiguates all memory references). We would like to know if the aliases really exist at run-time, and if any imprecision in the alias analysis causes missed opportunities for optimizations or other clients of the analysis. To detect imprecision and its impact, we also use a run-time limit analysis to determine missed optimization opportunities and their causes for a given program input. No previous work on alias analysis uses this metric.

The remainder of this section is organized as follows. Sections 3.1 and 3.2 describe our experimental framework and benchmark programs. Section 3.3 presents the static alias pairs for our analyses. Section 3.4 presents the simulated run-time improvements due to our alias analysis for redundant load elimination. Section 3.5 evaluates the room for improvement in our analysis.

Table 4: Description of Benchmark Programs

Name	Lines	Instructions	% Heap loads	% Other loads	Description
format [23]	395	1,879,195	10	17	Text formatter
dformat [23]	602	1,442,541	9	19	Text formatter
write-pickle	654	1,614,437	13	16	Reads and writes an AST
k-tree[3]	726	50,297,517	10	21	Manages sequences using trees
slisp	1,645	11,462,791	27	9	Small lisp interpreter
pp	2,328	45,779,402	11	19	Pretty printer for Modula-3 programs
dom [24]	6,186				System for building distributed applications
postcard	8,214				Graphical mail reader
m2tom3	10,574	50,894,990	8	28	Converts Modula-2 code to Modula-3
m3cg	16,475	5,636,004	8	21	M3 v. 3.5.1 code generator + extensions

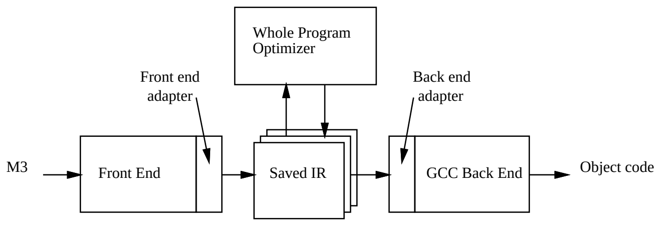

Figure 5: Compilation Framework

3.1 Environment

Figure 5 illustrates our compilation framework. The front end reads a Modula-3 module and generates a file containing a typed abstract syntax tree (AST) for the compiled module. The *whole program optimizer* (WPO) reads in the ASTs for a collection of modules, analyzes and transforms them, and then it writes out the modified AST for each module and a file with the corresponding low-level stack machine code. The stack representation is the input language for a back end based on GCC. WPO implements all optimizations and analyses presented in this paper.

3.2 Benchmarks

For each benchmark in our suite, Table 4 gives the number of non-comment, non-blank lines of code. For the non-interactive programs, Table 4 also gives the number of instructions executed, the percent of instructions that are memory loads from the heap, and the percent of instructions that are memory loads from the stack and global area (*other*). None of these programs were written to be benchmarks, but other researchers have used several of them in previous studies [16, 10]. Table 4 contains the data for the original programs (*i.e.,* without the optimizations proposed here) but with GCC's standard optimizations turned on, which include register allocation and instruction scheduling (except for `m2tom3`). Due to a compiler bug in GCC, we were unable to perform the standard optimizations on `m2tom3`, which explains its unusually large number of *other loads*. The numbers in Table 4 do not include instructions or memory references from the standard libraries.

3.3 Static Evaluation

Table 5 evaluates the relative importance of the three TBAA: *TypeDecl*: TBAA using only type declarations; *FieldTypeDecl*: TBAA using *TypeDecl* and field declarations; and *SMFieldTypeRefs*: TBAA using *FieldTypeDecl* and assignment statements. The *References* column gives the total number of heap memory references in the source of the benchmark programs. For each of the analyses, the table contains the number of local (*L Alias*) and global (*G Alias*) alias pairs. Local alias pairs are heap memory references within the same procedure that may alias each other, and global alias pairs are heap memory references not necessarily within the same procedure that may alias each other. Since each memory reference trivially aliases itself, we exclude this pair. Note that since *SMFieldTypeRefs* is strictly more powerful than *FieldTypeDecl*, and *FieldTypeDecl* is strictly more powerful than *TypeDecl*, we can use static metrics to compare the three.

From the table, we see that *TypeDecl* performs a lot worse than *FieldTypeDecl*, and that flow-insensitive merging using *SMFieldTypeRefs* offers little improvement over *FieldTypeDecl. SMFieldTypeRefs* improves local and global alias pairs on `postcard`, and the number of global aliases for `m3cg`. On average, each heap reference may alias 4.7 other intraprocedural references using *TypeDecl*, 3.4 references using *FieldTypeDecl*, and 3.4 references using *SMFieldTypeRefs.* The range is from 0.3 to 20.8 references for *SMFieldTypeRefs.* On average, each heap reference may alias 54.1 other interprocedural references using *TypeDecl*, 12.7 references using *FieldTypeDecl*, and 12.7 references using *SMFieldTypeRefs.* The range is from 2 to 27.7 references for *SMFieldTypeRefs.* The number of interprocedural aliases is much higher than the number of intraprocedural aliases, suggesting that TBAA is probably too imprecise for interprocedural optimizations. In the next two sections, we show that even though our analysis does not disambiguate all intraprocedural memory references (i.e., the local aliases are greater than zero), it may be precise enough for some applications.

3.4 Optimization Results

This section measures the static and simulated execution-time impact of TBAA on redundant load elimination (RLE). We first describe our implementation of RLE, and then show its impact on execution time. Section 3.5 then describes a limit analysis that demonstrates that with respect to RLE, there is

Table 5: Alias Pairs

Program	References	*TypeDecl* L Alias	*TypeDecl* G Alias	*FieldTypeDecl* L Alias	*FieldTypeDecl* G Alias	*SMFieldTypeRefs* L Alias	*SMFieldTypeRefs* G Alias
format	75	221	450	133	206	133	206
dformat	156	554	2665	293	1286	293	1286
write-pickle	171	383	2089	235	507	235	507
slisp	230	122	2322	74	464	74	464
pp	444	1626	10830	719	3811	719	3811
k-tree	612	2731	24344	1328	9655	1328	9655
dom	800	932	29550	589	21802	589	21802
m2tom3	904	19036	47856	18824	25048	18826	25048
postcard	1038	4208	30890	1623	5278	1615	5262
m3cg	4515	16521	1409449	6154	121476	6153	120525

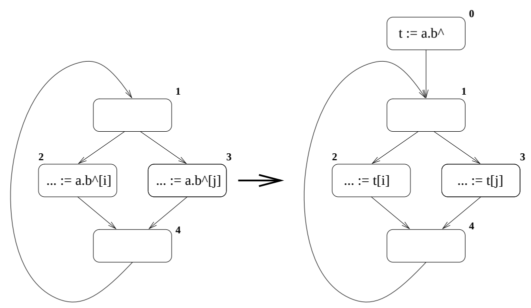

Figure 6: Eliminating Loop Invariant Memory Loads

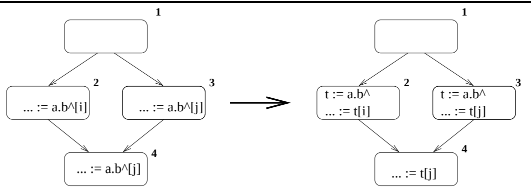

Figure 7: Eliminating Redundant Memory Loads

little or no room for improvement in TBAA.

3.4.1 Redundant Load Elimination

Redundant load elimination (RLE) combines variants of loop invariant code motion (similar to register promotion [8]) and common subexpression elimination [1], which most optimizing compilers perform. RLE differs from classic loop invariant code motion and common subexpression elimination in that it eliminates redundant loads instead of redundant computation. We expect RLE to be a profitable optimization since loads are expensive on modern machines and architects expect they will only get more expensive [18].

RLE hoists memory references out of loops if the reference is loop invariant and is executed on every iteration of the loop, leaving it up to the back end to place the hoisted memory reference in a register. For example in Figure 6, the access path `a.b^` is redundant on all paths, and loop invariant code motion moves it into the loop header. As shown in Figure 7, RLE also replaces redundant memory expressions by simple variable references, which the back end may place in a register. A memory expression at statement `s` is redundant if it is available on every path to `s`. RLE therefore improves performance by enabling the replacement of costly memory references with fast register references. Since RLE operates on memory references its effectiveness depends directly on the quality of the alias information (and also on the back end). To enable RLE across calls, RLE is preceded by a mod-ref analysis which summarizes the access paths that are referenced and modified by each call. For example, in order to hoist a memory reference out of a loop containing a call, TBAA needs to know whether the call changes the value of the memory reference. Note that even though RLE uses interprocedural mod-ref information, it does not eliminate redundant loads across procedure boundaries.

3.4.2 Impact of TBAA on RLE

Table 6 gives the number of access paths that RLE removes statically in each of our benchmark programs for each variant of TBAA: *TypeDecl*, *FieldTypeDecl*, and *SMFieldTypeRefs*, By comparing Table 6 and Table 5, we see that the differences between the number of local alias pairs is the strongest indicator of optimization opportunities for RLE. In particular, the big differences between the number of alias pairs for *TypeDecl* and *FieldTypeDecl* result in an increase in the number of redundant loads found by RLE. In contrast, the reductions in the number of alias pairs between *FieldTypeDecl* and *SMFieldTypeRefs* does not change the number of redundant loads found by RLE. (These reductions are however smaller than the others.)

Table 6: Number of Redundant Loads Removed Statically

Program	*TypeDecl*	*FieldTypeDecl*	*SMFieldTypeRefs*
format	27	29	29
dformat	10	22	22
write-pickle	46	47	47
k-tree	221	228	228
m2tom3	369	396	396
slisp	36	37	37
m3cg	524	613	613

We also measured execution times using a detailed (and validated [5]) simulator for an Alpha 21064 workstation with one difference: rather than simulating an 8K primary cache we simulated a 32K primary cache to eliminate variations due to conflict misses that we observed in an 8K direct mapped

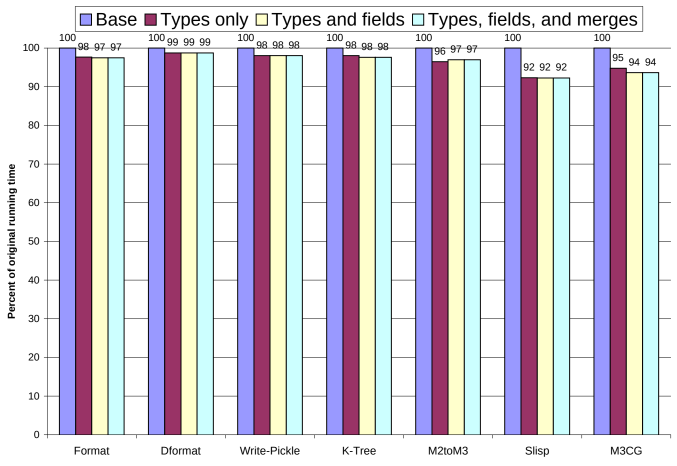

Figure 8: Impact of RLE

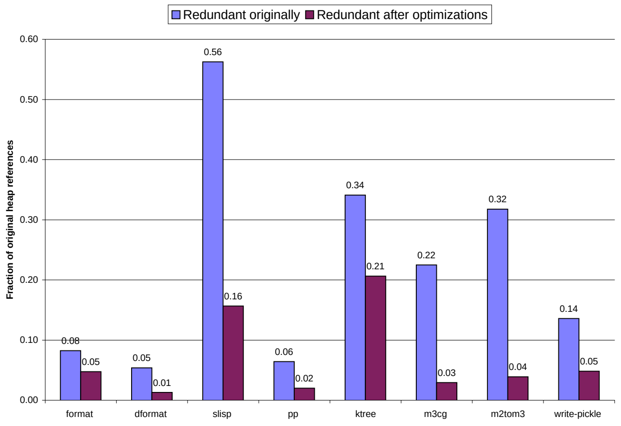

Figure 9: Comparing TBAA to an Upper Bound

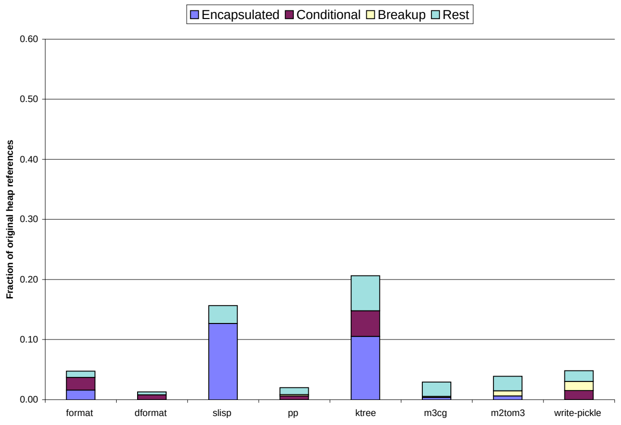

Figure 10: Source of Redundant Loads after Optimizations

cache. Also, we only measured the execution time spent in user code since that is the only code that we were able to analyze. Execution times are normalized with respect to the execution time of the original program without RLE, but with all of GCC's optimizations. (GCC eliminates redundant loads without any assignments to memory between them.)

Figure 8 illustrates the simulated execution time impact of TBAA on RLE relative to the original execution time. The graph has three bars for each non-interactive benchmark. Each bar represents the execution time due to RLE and a different alias analysis: *TypeDecl* (types only), *FieldTypeDecl* (types and fields), and *SMFieldTypeRefs* (types, fields, and merges).

TBAA enables RLE to improve program performance from 1% to 8%, and on average 4%. Since RLE is just one of many optimizations that benefits from alias analysis, the full impact of alias analysis on execution time should be higher. Also, contrary to what the data in Table 5 and Table 6 suggest, the three variants of TBAA have roughly the same performance *as far as* RLE *is concerned*. These results make two important points. First, a more precise alias analyses is not necessarily better; it all depends on how the alias analysis is used. Second, static metrics, such as alias pairs are insufficient by themselves for evaluating alias analyses.

3.5 Comparing TBAA to an Upper Bound

How much precision does TBAA *lose in order to achieve its fast time bound?* It is easy to contrive examples where TBAA fails to disambiguate memory references while many other alias analyses succeed. This section demonstrates, using a limit study, that for RLE and our benchmark programs, there is little to be gained from an alias analysis that is more precise than TBAA.

Figure 9 compares heap loads that are redundant at run time *before* and *after* applying RLE. A redundant load is when two consecutive loads of the same address load the same value in the same procedure activation. We measure these loads using *ATOM*[31], a binary rewriting tool for the Alpha. We instrument every load in an executable, recording its address and value. If the most recent previous load of an address is redundant with the current load, we mark it as redundant. (Elsewhere we describe this process in more detail [13].) In Figure 9, the black bars give the fraction of heap references that are redundant in the original program. The white bars give the fraction of heap references that are redundant after TBAA and RLE (this fraction is with respect to the original number of heap references). These results are specific to program inputs.

Figure 9 shows that our optimizations eliminate between 37% and 87% of the redundant loads in these programs. Moreover, for 6 of the 8 benchmark programs, only 5% or fewer of the remaining loads are redundant. However, `slisp` and `ktree` still have many redundant loads. To understand the source of all the remaining redundant loads, we manually classified them as follows:

1. **Encapsulation**: RLE could not eliminate a redundant expression because it was implicit in our high-level

(AST) intermediate representation. For example, the subscript expression for an open array involves an implicit memory reference to the dope vector.

2. **Conditional**: RLE did not eliminate a redundant expression because it was only partially redundant, *i.e.*, redundant along some paths but not along others. Partial redundancy elimination would catch these.
3. **Breakup**: RLE did not eliminate a redundant expression because it consisted of multiple smaller expressions and our optimizer does not do copy propagation.
4. **Alias failure**: TBAA did not disambiguate two memory references.
5. **Rest**: we don't know the reason why RLE did not eliminate the redundant loads since we did not determine the reason for the entire list of redundant expressions (which is labor intensive).

The first category is due to a limitation of representation, not TBAA or RLE. Categories 2 and 3 are limitations in our implementation of RLE, rather than TBAA. The fourth category, *alias failure*, corresponds to limitations of TBAA. The fifth category may be a limitation of RLE or TBAA or the representation. Each bar in Figure 10 breaks down the *Redundant after Optimizations* bar from Figure 9 into the above five categories.

Figure 10 illustrates that *Encapsulation* (dope vector accesses to index open arrays) is the most significant source of the remaining redundant loads. Figure 10 also shows that we did not encounter a single situation when optimization failed due to inadequacies in our alias analysis. Those redundant loads that could be due to failed analysis are categorized as *Rest*, and on average, are less than 2.5% of the remaining loads. Thus, for RLE on these programs and their inputs, there is not much room for improvement in our simple and fast alias analysis.

3.6 Summary of Results

This section evaluated TBAA using four different metrics:

- Number of static alias pairs.
- Run-time improvement due to an optimization that uses TBAA (RLE).
- Number of opportunities exposed by TBAA for RLE.
- An upper-bound for TBAA with respect to RLE.

Each of these four metrics exposes different information about TBAA. The first metric, *number of static alias pairs*, tells us two things. (1) For our benchmark programs, *SM-FieldTypeRefs* offers little or no precision over *FieldTypeDecl.* (2) *FieldTypeDecl* is potentially a much better alias analysis than *TypeDecl.* Even though *FieldTypeDecl* offers little performance improvement over *TypeDecl* for RLE, *FieldTypeDecl* should probably be the algorithm of choice since it does gives more precise results (without much added complexity) which may be important for other optimizations that use alias analysis.

The second metric, *run-time improvement*, indicates the how much an optimization or analysis really matters to the bottom line: performance. Our experiments find that the majority of the run-time improvement comes from *TypeDecl. FieldTypeDecl* improves performance only slightly. The results also illustrate that the run-time improvement due to our analysis and optimization is relatively small: on average 4% improvement. If run-time improvement is the only metric we use, then we might conclude that TBAA is a very imprecise alias analysis. However, *upper-bound analysis* reveals that TBAA in fact performs about as well as any alias analysis could perform with respect to RLE and our benchmarks programs.

The third metric, *number of opportunities exposed by* TBAA *for* RLE, reveals that *FieldTypeDecl* enables many more opportunities for RLE than *TypeDecl.* However, our run-time measurements find that *FieldTypeDecl* is only slightly better than *TypeDecl.* If we had used only run-time improvements to evaluate our analysis we might conclude that *TypeDecl* is the algorithm of choice. However, the *number of opportunities* metric tells us that *FieldTypeDecl* is indeed significantly better than *TypeDecl.* Perhaps with different benchmark inputs *FieldTypeDecl* may improve performance significantly more than *TypeDecl.*

Finally, the *upper-bound analysis for* RLE *using* TBAA reveals that a more precise alias analysis for RLE would yield few benefits: there is little or no room for improvement in TBAA with respect to RLE.

To summarize, the four metrics tell us different information about the different levels of TBAA. For this reason, we feel that *all* of these metrics should be used together in a thorough evaluation of an alias analysis (or for that matter any compiler analysis).

3.7 Cumulative Results

Figure 11 shows the cumulative impact of two sets of optimizations: method invocation resolution [14] plus inlining (*Minv + Inlining*) and RLE. Method resolution uses TBAA (and other analyses) to help resolve method invocations on object fields and array elements. While we expected method resolution and inlining to expose more opportunities for RLE, they did not. On studying the interactions of RLE with method invocations and inlining using limit analysis, we found that inlining exposes more redundant expressions but they are usually conditional (Section 3.5). Thus, while partial redundancy elimination can eliminate these redundant loads, RLE cannot. We plan to implement and evaluate partial redundancy elimination of memory expressions in future work.

4 Analyzing Incomplete Programs

Most prior pointer alias analyses for the heap are whole-program analyses, i.e., the compiler assumes it is analyzing the entire program, including libraries, making a *closed world assumption*. Many situations arise when the entire program is not available: for instance, during separate compilation, or compiling libraries without all their potential clients, or com-

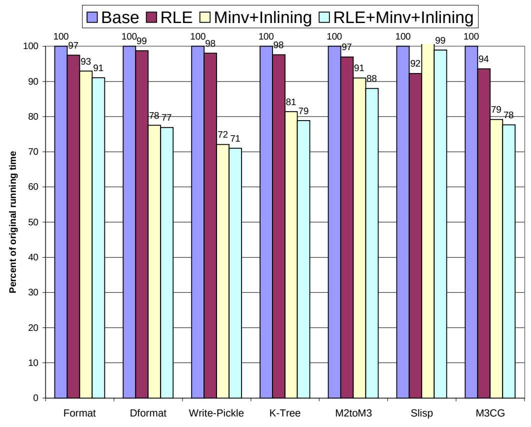

Figure 11: Cumulative Impact of Optimizations

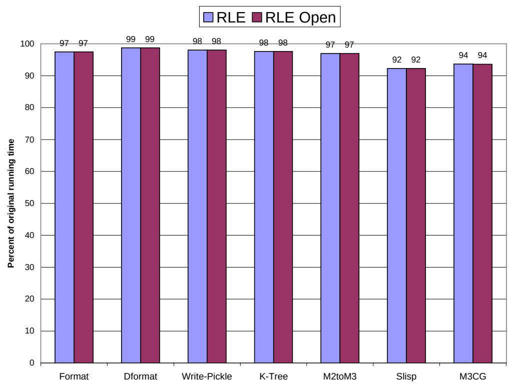

Figure 12: Open and Closed World Assumptions

piling incomplete programs.

In unsafe languages such as C++, alias analyses must assume that unavailable code may affect all pointers in arbitrary ways. For type-safe languages like Modula-3 and Java, the compiler can use type-safety and a type-based alias analysis to make stronger type-safety assumptions about unavailable code. It can assume that unavailable code will not violate the type system of the language. For example, consider the following procedure declaration using the types declared in Figure 1.

```
PROCEDURE f (p: S1; q: S2) = ...
```

In an unsafe language, if some of the callers of `f` are not available for analysis, the compiler must assume that `p` and `q` are aliases. For a type-safe language, a type-based analysis can safely assume that `p` and `q` are not aliases since they have incompatible types.

Two components of TBAA rely on properties other than the type system of the language: *AddressTaken* and type merging. Since unavailable code may pass the address of a qualified expression or subscript expression to available code we revise *AddressTaken* as follows.

AddressTaken (`p`) is true:

1. if the program ever takes `p`'s address (for instance to pass it by reference or as part of a `WITH`), or
2. if `f` is a pass-by-reference formal and `p` and `f` have the same type.

Since Modula-3 requires the types of pass-by-reference formals and actuals to be identical, the second clause needs to check only for type *equality*, not type *compatibility*. Note that this new definition of *AddressTaken* considers instructions in the program for available code (1) and considers only the type system for unavailable code (2).

Since unavailable code may cause merges of types, we make *SMFieldTypeRefs* more conservative at merges. We merge any two types (related by the subtype relation) to which it has access since unavailable code may assign them. Since Modula-3 uses structural type equivalence, unavailable code can access most types because it can construct its own copy of the types. Exceptions to this ability are *Branded* types in Modula-3. These types essentially observe name equivalence and may not be "reconstructed" by unavailable code.

Figure 12 compares the simulated run-time improvement due to redundant load elimination using TBAA when assuming that the entire program is available (closed world) and assuming it is not available (open world). Notice that in our experiments, the open-world assumption has an insignificant impact on the effectiveness of TBAA with respect to RLE. This result however reflects the results in Table 6, since *SMFieldTypeRefs*, which is most affected by the open world assumption, does not enable any additional opportunities for RLE over *FieldTypeDecl*. With respect to the static metrics, we found that they were the same for the open-world and closed-world assumptions with one difference: M3CG had about 80 more alias pairs (interprocedurally) with the open-world assumption than with the closed world assumption. However, the additional alias pairs did not reduce the effectiveness of RLE.

5 Related Work

Alias analysis must consider an unbounded number of paths through an unbounded collection of data, and is therefore harder than traditional data-flow analyses. The literature contains many algorithms for alias analysis [2, 4, 6, 7, 9, 15, 19, 8, 20, 21, 22, 30, 32, 35, 36]. The key differences between the algorithms stem from where and how they approximate the unbounded control paths and data. The approximation determines the precision and efficiency of the algorithm, and these alias analyses range from precise exponential time algorithms to less precise nearly linear time algorithms.

Our work differs from previous work in two ways: (1) It is type-based instead of instruction-based. (2) We evaluate our alias algorithm with respect to an optimization, redundant load elimination, and its upper bound, rather than using static measurements as used by most work on alias analysis [2, 4, 6, 7, 9, 15, 20, 21, 22, 30, 32, 35]. Our upper bound measurement is similar to Wall's [34], which assumes a "perfect alias analysis" to find an upper bound on instruction level parallelism. Wall [34] does not evaluate an existing alias analysis as we do, but just gives the potential of a perfect alias analysis for instruction level parallelism.

Aho, *et al.* [1] and Chase, *et al.* [6] were among the first to notice that using programming language types could improve alias analysis, but did not present algorithms that did so. Our alias algorithm is most similar to those of Rinard and Diniz [26], Steensgaard [32], and Ruf [27, 28].

Rinard and Diniz use type equality to disambiguate memory references. The type system they use is a subset of C++ that does not have inheritance and is thus weaker than Modula-3's or Java's type systems. Steensgaard uses an instruction-based alias algorithm which uses non-standard types, not programming language types, to obtain a fast alias analysis. His type inference algorithm is similar to our selective type merging; however, he does not use programming language types, and in particular inheritance, to prune the merge sets as we do. Ruf shows how to use programming language types to partition data-flow analyses: each partition represents code that can be analyzed independently and thus a different analysis can be used on each partition [28]. Ruf uses his scheme to partition programs for alias analyses, but does not use the programming language types in the analysis. Ruf [27] compares a context sensitive alias analysis to a context insensitive alias analysis and finds, for his benchmarks, that they are comparable in precision. Our work supports his in that we also find that a simple alias analysis can yield very precise results.

Cooper and Lu [8] describe and evaluate register promotion, an optimization that moves memory references out of loops and into registers. They evaluate register promotion with two alias analyses: a trivial analysis and a flow-sensitive alias analysis. They used the number of instructions executed as their performance metric and found that the more powerful alias analysis did not significantly improve performance. Our results support theirs: for many applications a fast and simple alias analysis may be sufficient.

Shapiro and Horwitz [29] evaluate the impact of three flow insensitive alias analyses on a range of optimizations. They evaluate their algorithms by counting optimization opportunities rather than any of the metrics that we use. They find that clients of alias analysis may run faster with a more precise alias analysis than with a less precise alias analysis. Similarly, Ghiya and Hendren [17] use pointer analysis to improve scalar optimizations, and present run-time improvements. This work was concurrent with ours, They do not present a limit study.

Debray *et al.* [11] describe an alias analysis for executable code. They evaluate their algorithm by measuring the percentage of loads eliminated by redundant load elimination. They do not present execution time improvements or a limit study for their alias analysis.

Since we ignore control flow, our algorithm achieves a $O(\text{Instructions} \times \text{Types})$ time complexity that is asymptotically as fast as the fastest existing alias analysis [32].

6 Conclusions

This paper describes and evaluates three algorithms that use programming language types to disambiguate memory references. The first analysis uses type compatibility to determine aliases. The second extends the first by using additional high-level information such as field names and types. The third, TBAA, extends the second with a flow-insensitive analysis. We show that the algorithm that uses only type compatibility is very imprecise whereas the other two analyses are much better at disambiguating memory references in the same procedure. We also evaluate TBAA with respect to redundant load elimination (RLE), one of its many potential clients. Our results show that TBAA and RLE improve program performance by up to 8%, and on average 4%. We demonstrate that with respect to RLE and these benchmark programs, TBAA is very precise; a more precise analysis could only enable RLE to eliminate on average an additional 2.5% of redundant references, and at most 6%. Because TBAA relies on type-safety, it can be conservative in the face of incomplete, type-safe programs without losing effectiveness. Our results show that as far as RLE is concerned, TBAA performs just as well with an open-world assumption as with a closed-world assumption.

TBAA achieves its fast time bound and accuracy because of type safety, and our results confirm a common (but to our knowledge, untested) belief that type safety can be used to improve program performance. Taken together, these results suggest that type-based alias analysis can be effective, and that a thorough evaluation of alias analyses with respect to their clients is necessary to understand their strengths and weaknesses.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] John Banning. An efficient way to find side effects of procedure calls and aliases of variables. In *Conference Record of the Sixth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 29–41, San Antonio, Texas, January 1979.

[3] Rodney M. Bates. K-trees. Personal communication, November 1994.

[4] Michael Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Efficient flow-insensitive alias analysis in the presence of pointers. Technical Report 19546, IBM T.J. Watson Research Center, Yorktown Heights, NY, September 1994.

[5] Brad Calder, Dirk Grunwald, and Joel Emer. A system level perspective on branch architecture performance. In *28th International Symposium on Microarchitecture*, pages 199–206, November 1995.

[6] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[7] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 232–245, Charleston, SC, January 1993.

[8] Keith Cooper and John Lu. Register promotion in c programs. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

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

[10] Jeffery Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the ACM SIGPLAN '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–100, San Jose, CA, October 1996.

[11] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Conference Record of the Twentyfifth Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1998.

[12] Digital Equipment Corporation. *DEC3000 300/400/500/600/800 Models: System Programmer's Manual*, 1 edition, September 1993. First Printing.

[13] Amer Diwan. *Understanding and improving the performance of modern programming languages*. PhD thesis, University of Massachusetts, Amherst, MA 01003, October 1996.

[14] Amer Diwan, Eliot Moss, and Kathryn S. McKinley. Simple and effective analysis of statically typed object-oriented programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Jose, CA, October 1996.

[15] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural Points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[16] Mary F. Fernandez. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 103–115, La Jolla, CA, June 1995. SIGPLAN, ACM Press.

[17] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Conference Record of the Twentyfifth Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1998.

[18] John Hennessy and David Patterson. *Computer Architecture A Quantitative Approach*. Morgan-Kaufmann, 1995.

[19] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 218–229, 1994.

[20] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem classification. In *Conference Record of the Eighteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 93–103, Orlando, FL, January 1991.

[21] William Landi and Barbara G. Ryder. Interprocedural side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, CA, June 1992.

[22] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, Atlanta, GA, June 1988.

[23] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[24] Farshad Nayeri, Benjamin Hurwitz, and Frank Manola. Generalizing dispatching in a distributed object system. In *Proceedings of European Conference on Object-Oriented Programming*, pages 450–473, Bologna, Italy, July 1994.

[25] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, New Jersey, 1991.

[26] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, June 1996.

[27] Eric Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, CA, June 1995.

[28] Erik Ruf. Partitioning dataflow analyses using types. In *popl97*, Paris, France, January 1997.

[29] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In Pascal Van Hentenryck, editor, *Lecture Notes in Computer Science, 1302*, pages 16–34. Springer-Verlag, 1997. Proceedings from the *4th International Static Analysis Symposium*.

[30] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[31] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, June 1994. Association of Computing Machinery.

[32] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twentythird Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 32–41. Association of Computing Machinery, January 1996.

[33] Sun Microsystems Computer Corporation. *The Java language specification*, 1.0 beta edition, October 1995.

[34] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on*

Architectural Support for Programming Languages and Operating Systems, pages 176–189, Santa Clara, California, 1991.

[35] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the Seventh Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 83–94, Las Vegas, Nevada, January 1980.

[36] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, La Jolla, CA, June 1995. Association of Computing Machinery.