# Using Types to Analyze and Optimize Object-Oriented Programs

AMER DIWAN
University of Colorado, Boulder

and
KATHRYN S. MCKINLEY and J. ELIOT B. MOSS
University of Massachusetts, Amherst

Object-oriented programming languages provide many software engineering benefits, but these often come at a performance cost. In this paper, we address two of these costs: method invocations and pointer dereferencing. We show how to use types to produce effective, yet simple techniques that reduce the costs of these features in Modula-3, a statically typed, object-oriented language. Our compiler performs alias analysis to disambiguate memory references and uses it to eliminate redundant memory references. It also identifies method-invocation sites that are monomorphic, replacing them with direct calls. Using limit, static, and run-time evaluation, we demonstrate that these techniques are effective, and often close to perfect for a set of Modula-3 benchmarks. By reducing these costs, we take a step towards making object-oriented languages faster and thus even more appealing.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Algorithms, Languages, Performance, Measurement

Additional Key Words and Phrases: alias analysis, polymorphism, classes and objects, object orientation, method invocation, redundancy elimination

## 1. INTRODUCTION

In object-oriented languages, programmers use pointers, type hierarchies, and method invocations to improve code reuse and correctness. These features have a cost. For example, without an alias analysis, the compiler must assume all pointer dereferences are potential aliases and may not reorder them. Compilers need to reorder

Authors' address: Amer Diwan, Department of Computer Science, University of Colorado, Boulder, CO 80309. Kathryn S. McKinley and J. Eliot B. Moss, Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610. Internet addresses: diwan@cs.colorado.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 Compaq. 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. Any opinions, findings, and conclusions or recommendations expressed in this material are the authors and do not necessarily reflect those of the sponsors.
Early versions of some of the results in this paper have been published in OOPSLA 95 and PLDI 98.

instructions to exploit effectively the underlying hardware which may have multiple functional units and elaborate pipelines. An effective alias analysis disambiguates memory references, and enables the compiler to reorder statements that do pointer accesses.

Method invocations are costly as well. Method invocations obscure which procedure is actually being invoked. In dynamically-typed languages, frequent method look-up is costly in itself [Chambers 1992], but in statically-typed languages, it is typically not a significant cost. For both static and dynamic languages however, method invocations inhibit optimization. If analysis can resolve method invocations to direct calls, the compiler can replace the method invocation with a direct call, a tailored call, or an inlined call. The additional control-flow information provides fodder for an optimizing compiler to improve performance.

To alleviate the performance degradation due to pointer references we present a range of *type-based* alias analyses. Type-based analyses (TBAA) use or operate on programming-language types. Our alias analyses techniques range from a simple inspection of the type hierarchy to interprocedural flow-insensitive analysis. We determine the effectiveness of our alias analyses with respect to two optimizations: *redundant load elimination* (RLE) and method resolution. RLE combines loop invariant code motion and common subexpression elimination of memory references. To alleviate the performance degradation of method invocations, we also use a range of type-based techniques. These techniques range from a simple inspection of the type-hierarchy to interprocedural flow-sensitive context-insensitive analysis. Our more aggressive method resolution techniques use TBAA. While there are obvious interactions between pointer analysis and method resolution, we adopted a fixed order between the analyses, but this process could clearly be iterative. A few of our alias analyses and method resolution techniques have been proposed or implemented before in the literature. However, our evaluation reveals new insights about these techniques.

For both alias analysis and method invocation resolution, we evaluate the range of algorithms with static, dynamic, and limit analyses. Our results show that these simple techniques are surprisingly close to perfect, and that the simplest inspection of the type hierarchy can yield dramatic improvements in precision. Although others have proposed using types in similar ways, we believe we are the first to demonstrate their unanticipated effectiveness for two important optimizations: RLE and method resolution. We also modify our analyses to work on incomplete programs and demonstrate that their effectiveness is not compromised in many cases. Although we have implemented our analyses in a traditional optimizing compiler for Modula-3, the speed and simplicity of these analyses makes them practical for use in just-in-time compilers for languages like Java.

The remainder of this paper is organized as follows. Section 2 describes our type-based alias analysis algorithms. It begins with a brief technical background and then discusses three progressively more precise alias analyses based on type declarations, field declarations and other high-level properties, and finally flow-insensitive data-flow analysis. Section 3 begins also with a brief technical background, and then describes methods for resolving method invocations: type hierarchy analysis, intra and interprocedural type propagation, and intra and interprocedural type propagation using type-based alias analysis. Section 4 presents static, dynamic, and upper

Table I. 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` |

bound evaluation of the alias analysis and method resolution algorithms. Section 5 presents the cumulative results of applying both sets of analyses and optimizations presented in this paper. Section 6 extends and evaluates these algorithms for incomplete programs. Section 7 considers how our techniques apply to other object-oriented languages, particularly C++ and Java. Section 8 discusses related work in alias analysis and method resolution. Section 9 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 [Nelson 1991] or Java [Sun Microsystems Computer Corporation 1995] that does not support arbitrary pointer type casting which 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 I lists the three kinds of memory references in Modula-3 programs, their names, and a short description of each.[1]

We call a non-empty string of memory references, for example `a^.b[i].c`, an *access path* ($\mathcal{AP}$) [Larus and Hilfinger 1988]. 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.

### 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:

$$\mathit{Subtypes}\ (\mathit{Type}\ (\mathtt{p})) \cap \mathit{Subtypes}\ (\mathit{Type}\ (\mathtt{q})) \neq \emptyset.$$

[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;
```

Fig. 1. Type Hierarchy Example

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

$$\begin{aligned} &\mathit{Subtypes}\ (\mathit{Type}\ (\texttt{t})) \cap \mathit{Subtypes}\ (\mathit{Type}\ (\texttt{s})) \neq \emptyset \\ &\mathit{Subtypes}\ (\mathit{Type}\ (\texttt{t})) \cap \mathit{Subtypes}\ (\mathit{Type}\ (\texttt{u})) \neq \emptyset \\ &\mathit{Subtypes}\ (\mathit{Type}\ (\texttt{s})) \cap \mathit{Subtypes}\ (\mathit{Type}\ (\texttt{u})) = \emptyset \end{aligned}$$

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

### 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 TBAA *FieldTypeDecl*. It distinguishes accesses such as `t.f` and `t.g`, `f` $\neq$ `g`, that *TypeDecl* misses. The *FieldTypeDecl* algorithm appears in Table II. 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 tem-

Table II. *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`) $\wedge$ *FieldTypeDecl* (`p`, `q`) |
| 3 | `p.f` | `q^` | AddressTaken (`p.f`) $\wedge$ *TypeDecl* (`p.f`, `q^`) |
| 4 | `p^` | `q[i]` | AddressTaken(`q[i]`) $\wedge$ *TypeDecl* (`p^`, `q[i]`) |
| 5 | `p.f` | `q[i]` | false |
| 6 | `p[i]` | `q[j]` | *FieldTypeDecl* (`p`, `q`) |
| 7 | `p` | `q` | *TypeDecl* (`p`, `q`) |

porary 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 II 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 refer to 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.

The Java programming language will have similar rules. For C++ 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 refer to 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 refer to the same location because it is semantically correct for objects of type `T` to refer to objects of type `S1`. By inspecting the code however, it is obvious that `t` and `s` never refer to 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 each 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 each type t do
  let g ∈ Group, t ∈ g
    TypeRefsTable (t) = g ∩ Subtypes (t)
```

Fig. 2. Selective Type Merging

Figure 2 presents the algorithm to merge types selectively.[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 $\mathcal{AP}$s p and q, *SMTypeRefs* (p,q) indicates that they may be aliases if and only if:

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

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 refer to 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 refer to any object of type T1 or T2.

Step 1 initializes Group such that each declared type is in an independent set. An $\mathcal{AP}$ declared with type T is thus assumed to refer to 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 refer to objects of type T or type S1, but an $\mathcal{AP}$ declared as S1 may not refer to 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 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

[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 [Steensgaard 1996].

[4] If we took Steensgaard's algorithm [Steensgaard 1996] 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;
```

Fig. 3. Example to Illustrate *SMTypeRefs*

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 III. Notice that *SMTypeRefs* determines $\mathcal{AP}$s declared to be of type `T` may not refer to objects of type `S3`, but *TypeDecl* assumes they may.

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

### 2.5 Complexity of analyses

The complexity of the slowest TBAA (*SMFieldTypeRefs* and *FieldTypeDecl*) 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. For instance, computing all the *may-alias* pairs using TBAA (or any other *points-to* analysis) takes $O(e^2)$ time, where $e$ is the number of memory expressions in the program.

### 2.6 Using TBAA

Most compiler analyses and optimizations can benefit from alias analysis. In this section, we describe a sample optimization, redundant load elimination (RLE), that uses TBAA. We will later use RLE to evaluate TBAA. In Section 3.4 we describe another analysis that benefits from TBAA.

RLE combines variants of loop invariant code motion (similar to register promotion [Cooper and Lu 1997]) and common subexpression elimination [Aho et al.

Table III. *TypeRefsTable* for Figure 3

| Type | *TypeRefsTable* (Type) |
|---|---|
| T | `T`, `S1`, `S2` |
| S1 | `S1` |
| S2 | `S2` |
| S3 | `S3` |

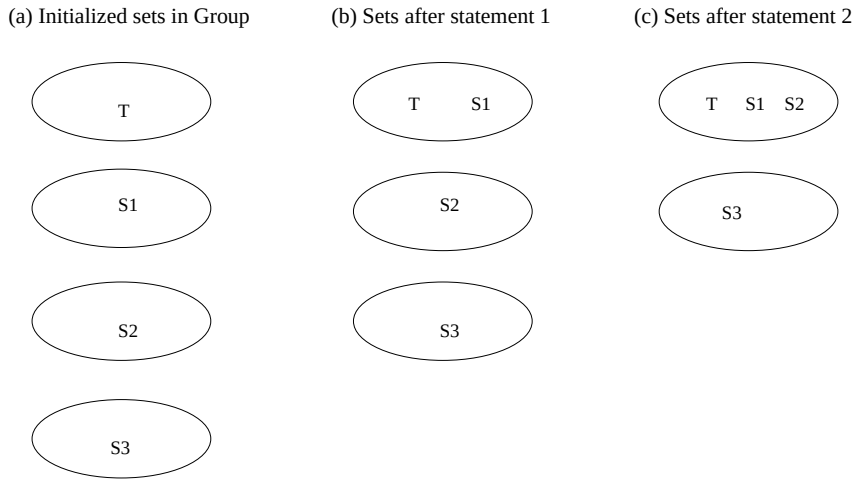


Fig. 4. Selective Merging for Figure 3

1986], 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 [Hennessy and Patterson 1995].

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 5, 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 6, 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 that 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. RESOLVING METHOD INVOCATIONS

In this section, we first describe polymorphic method invocations and discuss their implications. Then we describe five analysis techniques that resolve method invocations: (1) type hierarchy analysis, (2) intraprocedural type propagation, (3) intraprocedural type propagation using TBAA (4) interprocedural type propagation,

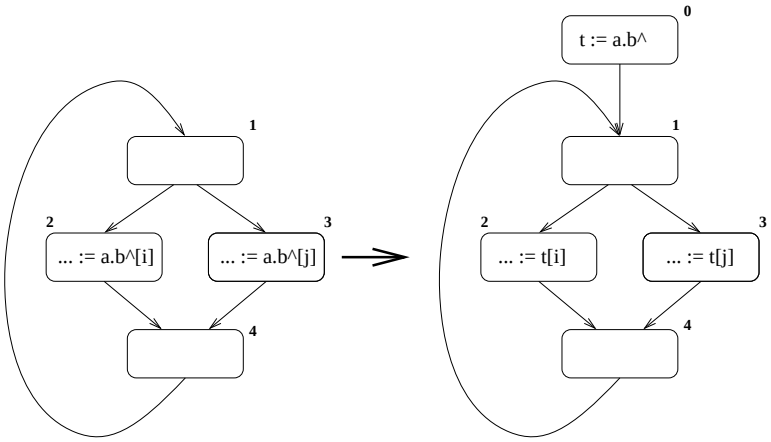


Fig. 5. Eliminating Loop Invariant Memory Loads

and (5) interprocedural type propagation using TBAA. We use the type hierarchy of Figure 7 as a running example to illustrate the strengths and limitations of the analyses. We use a power set of the types for the lattice for our analyses; the initial type for a variable or heap or record field or pointer reference is the empty set.

### 3.1 Background: Polymorphism through subtyping

Statically-typed object-oriented languages support polymorphism through subtyping. A type `S` is a subtype of `T` if it supports all the behavior of `T`. Thus, the program can use an object of type `S` whenever an object of type `T` is expected. In particular, a variable with *declared type* `T` may refer to objects that are subtypes of `T`, not just `T`.

Consider the Modula-3 type hierarchy in Figure 7, which defines a type `T`, and `S`, a subtype of `T`. `S` has all the behavior of `T` (in particular, the `m` method) but it may have different implementations of `T`'s methods (in this case, `mS` instead of `mT`). `S` may support behavior not supported by `T` (in this example, the `n` method). Invoking the `m` method on a variable with declared type `T` may invoke one of three procedures:

(1) `mT`, if the last object assigned to the variable had type `T`;

(2) `mS`, if the last object assigned had type `S`; and

(3) `error`, if the last object assigned was `NIL`.

In general, invoking a method on a variable (the *receiver*) can call any procedure that overrides that method in the variable's declared type or any subtype of its declared type. The `NULL` type, which contains a single value NIL, is a subtype of all objects in Modula-3 and overrides all methods with an `error` procedure.

A *polymorphic* method invocation site calls more than one user procedure at run time. For example, consider invoking the `print` method on each element of a linked list in a loop. If the list links objects of different types, then the `print` method invokes different procedures depending on the type of the list element.

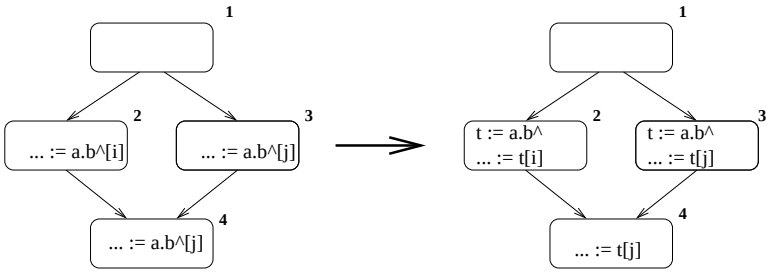


Fig. 6. Eliminating Redundant Memory Loads

A *monomorphic* method invocation site always invokes the same user procedure (or `error`). It may have the potential to be polymorphic but the polymorphism is never present at run time. To continue the linked list example, if the list links objects of only one type, then the `print` method will always invoke the same procedure.

A method invocation is *resolved* if it is identified as being monomorphic. The goal of the analyses presented here is to resolve all the monomorphic sites. We then use this information to replace method invocations with direct calls. Other work on dynamic languages converts important polymorphic method invocations to direct calls using *splitting*, which duplicates code in order to improve type information [Chambers and Ungar 1990]. Splitting works for both polymorphic and monomorphic method invocations. We instead focus on analysis, but the compiler could apply transformations for unanalyzable sites in a later phase.

### 3.2 Type Hierarchy Analysis

Type hierarchy analysis bounds the set of procedures a method invocation may call by examining the type hierarchy declarations for method overrides. For each type `T` and each method `m` in `T`, type hierarchy analysis finds all overrides of `m` in the type hierarchy rooted at `T`. These overrides are the procedures that may be called

```
TYPE T = OBJECT
           f: T;
         METHODS
           m := mT;
         END;
(* S is a subtype of T *)
TYPE S = T OBJECT
         METHODS
           n := nS;
         OVERRIDES
           m := mS;
         END;
```

Fig. 7. A Modula-3 Type Hierarchy

when m is invoked on a variable of type T. Since NULL is a subtype of all objects in Modula-3 and it overrides all methods, type hierarchy analysis can never narrow down the possibilities to just one; at best it determines a method is one procedure or the error procedure.

Type hierarchy analysis does not examine what the program actually does, just its type and method declarations. Thus, it takes time proportional to the number of types and methods in the program, $O(|Types| * |Methods|)$.

### 3.3 Intraprocedural Type Propagation

Intraprocedural type propagation improves the results of type hierarchy analysis by using data flow analysis to propagate types from *type events* to method invocations within a procedure. Type events create or change type information. The three distinguishing type events are allocation (v ← *NEW* (t)), implicit and explicit type discrimination operators (*IsType* (v, T)), and assignment (v ← u), which includes parameter bindings at calls.

For example, consider the following code:

```
1  p := NEW (S);
   IF cond THEN
2    o := NEW (T);
3    o.m ();
   ELSE
4    o := p;
5    o.m ();
   END;
6  o.m ();
```

Statement 2 contains a type event: an allocation. The allocation propagates the type T to o, and thus determines that the method invocation in statement 3 calls procedure mT. Similarly, statement 4 contains a type event: an assignment. The type event propagates the type of p to o, and thus determines that the method invocation in statement 5 calls procedure mS. Finally, intraprocedural type propagation merges the types of o at the control merge before statement 6, yielding the type {T, S}. Thus the method invocation in statement 6 cannot be resolved to a single procedure.

Our implementation of type propagation propagates types only to scalars; it assumes the conservative worst case (the declared type) for the allocated types of record fields, object fields, array references, and pointer accesses.

We formulate intraprocedural type propagation as a data-flow problem similar to reaching definitions. We identify and propagate pairs of variables and sets of possible types for the variables. All variables initially have the empty type. A statement $s$ with a type event generates and kills types as follows:

$$\begin{aligned}
\textsc{GenType}\ (v \leftarrow \mathit{NEW}(t)) &= \langle v, t\rangle \\
\textsc{GenType}\ (\mathit{IsType}(v, T)) &= \langle v,\ \mathit{TypeOf}(v) \cap T\rangle \\
\textsc{GenType}\ (v \leftarrow u) &= \langle v,\ \mathit{TypeOf}(u)\rangle
\end{aligned}$$

$$\begin{aligned}
\textsc{KillType}(v \leftarrow \mathit{NEW}(t)) &= \langle v,\ \mathit{TypeOf}(v)\rangle \\
\textsc{KillType}(v \leftarrow u) &= \langle v,\ \mathit{TypeOf}(v)\rangle
\end{aligned}$$

$T$ denotes a set of types and $t$ denotes a single type. *TypeOf* returns the current types of a variable. *IsType* is an explicit type discrimination event that checks if $v$'s type is in $T$. Type discrimination may also be implicit. In particular, for each *IsType*, there is an implicit type discrimination event for the false branch.

The data-flow equations for a statement $s$ are similar to the equations for reaching definitions:

$$\begin{aligned} \text{IN}(s) &= \bigcup_{p \in \text{PRED}(s)} \text{OUT}(p) \\ \text{OUT}(s) &= \text{GENTYPE}(s) \cup (\text{IN}(s) - \text{KILLTYPE}(s)) \end{aligned}$$

Our implementation stores the possible types of a variable as a set. Thus, the union and intersection operators are set union and set intersection respectively. This problem formulation is monotone and distributive.[5] Since Modula-3 programs are always reducible and type propagation is *rapid* [Kam and Ullman 1976], we use an $O(n * v)$ solution, where $n$ is the number of statements in a procedure, $v$ is the number of variables in the procedure, and each step of the algorithm is a bit vector operation.

### 3.4 Intraprocedural type propagation with TBAA

Intraprocedural type propagation assumes worst case about pointer dereferences. In this section, we show how to extend intraprocedural type propagation with TBAA (more specifically *SMFieldTypeRefs*) and thus improve the precision of intraprocedural type propagation with respect to pointer dereferences.

The main idea is simple: whenever intraprocedural type propagation enounters a pointer dereference, it invokes *SMFieldTypeRefs* to get the set of locations referenced by the pointer dereference. *SMFieldTypeRefs* summarizes this set compactly using type information (e.g., field $f$ of object type $O$). Intraprocedural type propagation can then propagate the types to or from the set of locations referenced by the pointer dereference. For example,

```
v: T;
v^.f := <rhs>
```

propagates the types of `<rhs>` to the field `f` of all possible aliases of $v$. In the worst case, this assignment propagates the type of the `<rhs>` to field $f$ of all subtypes of $T$ plus other variables if the program ever takes the address of an $f$ field (see Section 2.3). Since *SMFieldTypeRefs* computes *may points to* rather than *must points to* information, the analysis assumes that the aliases of $v.f$ may either retain their old type or get a new type from `<rhs>`. This is called a weak update in the pointer analysis literature.

This analysis discovers monomorphic uses of general data structures. Consider the linked list package again. This analysis detects when a program links objects of a single type and thus would resolve the invocation of the `print` method on the list elements. However, if the program allocates two distinct linked lists of the same type, but one with elements of type `S` and the other with type `T`, this analysis does not recognize that each list is homogeneous. It infers the type {`T, S, NULL`} for the elements in both lists.

[5] With the type discrimination operations in Modula-3, a more precise, but non-monotone formulation is possible. As far as we know, no one has investigated this formulation.

The type of an object-typed field always includes `NULL` since all fields in Modula-3 are initialized at allocation, and thus the first assignment to every object-typed field is always `NIL`.

The complexity of intraprocedural type propagation using TBAA is $O(n * (v + NT * NF))$, where $n$ is the number of statements in a procedure, $v$ is the number of variables in the procedure, $NT$ is the number of types in the program, $NF$ is the maximum number of fields in any type in the program, and each step of the algorithm is a bit vector operation.

### 3.5 Interprocedural Type Propagation

Interprocedural type propagation goes beyond intraprocedural type propagation to resolve more method invocations. It begins by building a call graph of the program. The call graph has an edge from a method invocation to each possible target determined by the earlier intraprocedural analysis. The algorithm operates by maintaining a work list of procedures that need to be analyzed. A procedure needs to be analyzed if new information becomes available about its parameters or about the return value of one of its callees. When interprocedural type propagation analyzes a procedure, it may put the callers and callees of the procedure on the work list and update the call graph. In particular, analysis may eliminate some call graph edges if it refines the type of a method receiver. Interprocedural type propagation maintains the work list in depth first order and terminates when the work list is empty.

Interprocedural type propagation also keeps track of which procedures are called only via method invocations (*i.e.*, not called directly). For these procedures, it eliminates `NULL` as a possible type for the first argument (`self`). (If `self` is `NIL`, then `error` is invoked instead of this procedure.)

Interprocedural type propagation propagates types only to scalars, and it assumes the most conservative type (the declared type) for all data accessed through pointer traversal. Interprocedural type propagation does not propagate side effects from calls and assigns the most conservative type for any variable changed by the call: the declared type. Variables potentially changed by a call include variables declared in outer scopes, globals, parameters passed by reference, and parameter aliases.

A distinguishing characteristic of our interprocedural analysis is that, unlike related work ([Agesen and Hölzle 1995; Plevyak and Chien 1994; Pande and Ryder 1995]), our analysis is context insensitive. Rather than analyzing for every combination of call site and callee, we merge the parameter types of all call sites of a procedure, and the return types of all callees at a call site. This simplification yields a faster analysis (quadratic instead of exponential) but at the cost of some accuracy. Consider the following code:

| Analysis | Eliminates NULL | Complexity |
|---|---|---|
| Type Hierarchy | No | $O(\|Types\| * \|Methods\|)$ |
| Intraprocedural Type Propagation | Yes | $O(\sum_p n_p * v_p)$ |
| Intraprocedural Type Propagation using TBAA | Yes | $O(\sum_p (n_p + NT * NF))$ |
| Interprocedural Type Propagation | Yes | $O(p \sum_p n_p * v_p)$ |
| Interprocedural Type Propagation using TBAA | Yes | $O(p \sum_p (n_p + NT * NF))$ |

Table IV. Summary of analyses

```
PROCEDURE Caller1 () =
  t := P (NEW (T));
  t.m ();

PROCEDURE Caller2 () =
  t := P (NEW (U);
  t.m ();

PROCEDURE P (o: T): T =
  RETURN o;
```

A context-sensitive analysis would analyze `P` separately for each of its call sites and thus determine that the method invocation in `Caller1` will call `mT` and that in `Caller2` will call `mU`. Our context-insensitive analysis instead merges the parameter types for each caller of `P` and thus does not resolve the method invocations in `Caller1` and `Caller2`. We show in Section 4.5.2 that for our benchmark suite, this loss in precision is not significant.

If interprocedural type propagation is invoked on an incomplete program, it makes conservative assumptions about the parameters of procedures that could be called from unavailable code, and about return values of unavailable procedures.

We can extend interprocedural type propagation with TBAA in the same way we extended intraprocedural type propagation with TBAA.

Since interprocedural type propagation may analyze each procedure multiple times (due to recursive or potentially recursive procedures), it may be substantially slower than intraprocedural type propagation. Since information flows forward through parameters and backward from return values, the worst case complexity is $O(|\text{procedures}| * \sum_p^{procedures} n_p * v_p)$, where $n$ is the number of statements in procedure $p$ and $v_p$ the number of variables in procedure $p$. In practice, however, we have found it to be linear in the number of instructions, analyzing each procedure an average of 2 to 4 times. The complexity of interprocedural type propagation using TBAA is similar, except that now we are propagating types not just to variables but also to aliases which are represented by types and fields in types.

### 3.6 Summary and complexity of analyses

Table IV summarizes the analyses. *Eliminates* `NULL` indicates whether the analysis can eliminate `NULL` as a possible type. In the *Complexity* column $n_p$ is the number of statements in procedure $p$, $NT$ is the number of types in the program and $NF$ is the maximum number of fields in any type. These algorithms are simple and

therefore fast, as shown in the *Complexity* column.

### 3.7 Using method resolution analyses

We use the results of method resolution to replace method invocations with direct calls, to replace method invocations with inlined calls, and to improve the precision of MOD-REF information for RLE.

## 4. RESULTS

This section evaluates type-based alias analysis and method resolution analyses using static and dynamic metrics, and a *limit* analysis. The majority of previous work on alias analysis [Banning 1979; Burke et al. 1994; Chatterjee et al. 1999; Chase et al. 1990; Choi et al. 1993; Cooper and Kennedy 1989; Deutsch 1994; Emami et al. 1994; Landi and Ryder 1991; 1992; Larus and Hilfinger 1988; Shapiro and Horwitz 1997b; Steensgaard 1996; Weihl 1980] evaluates alias analysis using only static properties, such as the sizes of the *may alias* and *points-to* sets. A few researchers recently have used dynamic evaluation such as measuring the *execution-time improvement* due to an optimization that uses alias analysis [Hummel et al. 1994; Wilson and Lam 1995; Cooper and Lu 1997; Ghiya and Hendren 1998; Shapiro and Horwitz 1997a]. The majority of previous work on method invocation resolution has evaluated the analyses using dynamic and static evaluation [Agesen 1995; Burke et al. 1995; Bacon and Sweeney 1996; Calder and Grunwald 1994; Carini et al. 1995; Dean et al. 1994; Fernandez 1995; Oxhoj et al. 1992; Pande 1996; Plevyak and Chien 1994; Pande and Ryder 1995; Palsberg and Schwartzbach 1991]. Bacon et al. [Bacon and Sweeney 1996] in concurrent work has used limit evaluation to evaluate method resolution analyses.

We first review the strengths and weaknesses of static, dynamic, and limit evaluations, then we describe our benchmarks and evaluation envionment, and finally, we present the evaluation for our alias and method resolution analyses.

**Static Evaluation.** Static properties, such as the size of the may-alias sets, enable comparisons between the precision of two similar analyses. Static properties have, however, two main disadvantages. (1) They 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. Similarly, even if an analysis resolves most method invocations, it may not improve performance substantially. (2) Static properties do not enable comparisons between the *effectiveness* of two analyses with different strengths and weaknesses. For example, the size of the alias sets of two analyses may be the same, but the analyses may disambiguate different pointers. Similarly two analyses may resolve the same number of method invocations in a program, but they may resolve different method invocations.

**Dynamic Evaluation.** Using dynamic evaluation, such as execution-time improvement, complements static metrics, since execution-time improvements measure the impact of the alias analysis or method resolution on its clients (usually compiler optimizations) directly. 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 im-

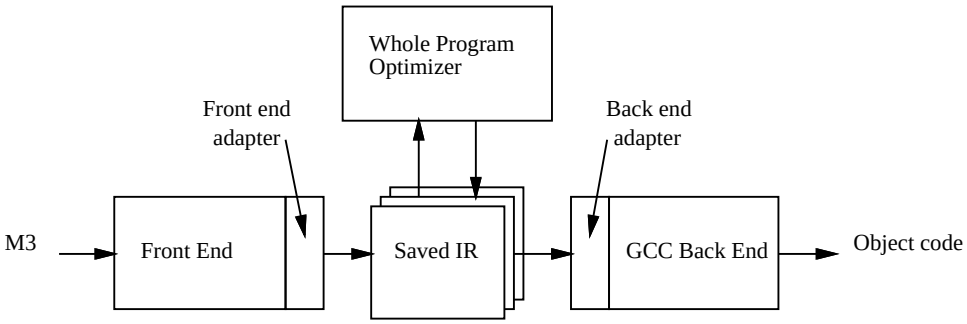


Fig. 8. Compilation Framework

provement there is in the analysis being evaluated (except in the unusual case of an alias analysis that disambiguates all memory references, or a method resolution analysis that resolves all method invocations). 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. Similarly, we would like to know if an unresolved method invocation is actually polymorphic at run-time and if not, why our analyses were unsuccessful. To detect such 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. Bacon and Sweeney [Bacon and Sweeney 1996] use a limit evaluation to find the imprecision in their method resolution analyses but do not use this technique to find the reason for the imprecision.

### 4.1 Compiler Framework

Figure 8 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 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 GCC [Stallman ] back end. WPO implements all optimizations and analyses presented in this paper.

### 4.2 Benchmarks

For each benchmark in our suite, Table V describes the benchmark and Table VI gives the number of non-comment, non-blank lines of code, the time to compile and link the benchmark on a 350 MHz Alpha 21164 workstation, and the number of method invocations at compile time and at run-time. For the non-interactive programs, Table VI 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 [Fernandez 1995; Dean et al. 1996]. Table VI contains

Table V. Description of Benchmark Programs

| Name | Description |
|---|---|
| format | Text formatter [Liskov and Guttag 1986] |
| dformat | Text formatter [Liskov and Guttag 1986] |
| write-pickle | Reads and writes an AST |
| k-tree | Manages sequences using trees [Bates 1994] |
| slisp | Small lisp interpreter |
| dom | System for building distributed applications [Nayeri et al. 1994] |
| postcard | Graphical mail reader |
| m2tom3 | Converts Modula-2 code to Modula-3 |
| m3cg | M3 v. 3.5.1 code generator + extensions |
| trestle | Window system + small application |

the data on 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. 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 VI do not include instructions or memory references from the standard libraries.

### 4.3 Ordering the analyses

In this work, we start by building the call graph using type hierarchy analysis, apply the alias analysis, apply method resolution analyses (such as interprocedural type propagation) and related transformations, and finally do RLE. There are however interactions between call graph building, method resolution, and alias analysis, and this process could clearly be iterative. Exploring the interactions between these analyses is beyond the scope of this paper.

### 4.4 Evaluation of TBAA

Sections 4.4.1, 4.4.2, and 4.4.3 evaluate TBAA using static, dynamic, and limit evaluations respectively.

Table VI. Statistics of Benchmark Programs

| Name | Lines | Build Time (sec) | Instructions | % Loads | | Method inv. | |
|---|---|---|---|---|---|---|---|
| | | | | Heap | Other | Static | Dynamic |
| format | 395 | 33.1 | 1,879,195 | 10 | 17 | 37 | 47,064 |
| dformat | 602 | 16.3 | 1,442,541 | 9 | 19 | 95 | 30,775 |
| write-pickle | 654 | 33.0 | 1,614,437 | 13 | 16 | 19 | 21,251 |
| k-tree | 726 | 26.7 | 50,297,517 | 10 | 21 | 13 | 714,619 |
| slisp | 1,645 | 23.8 | 11,462,791 | 27 | 9 | 223 | 67,253 |
| dom | 6,186 | 38.9 | (interactive) | | | 222 | |
| postcard | 8,214 | 67.8 | (interactive) | | | 293 | |
| m2tom3 | 10,574 | 280.9 | 50,894,990 | 8 | 28 | 1821 | 473,559 |
| m3cg | 16,475 | 339.4 | 5,636,004 | 8 | 21 | 1808 | 32,850 |
| trestle | 28,977 | 433.7 | (interactive) | | | 430 | |

Table VII. Static alias pairs as a percent of all possible pairs

| Program | % *TypeDecl* | Intraprocedural *FieldTypeDecl* | % *SMFieldTypeRefs* | % *TypeDecl* | Interprocedural *FieldTypeDecl* | % *SMFieldTypeRefs* |
|---|---|---|---|---|---|---|
| format | 31 | 27 | 27 | 11 | 8 | 8 |
| dformat | 24 | 16 | 16 | 19 | 11 | 11 |
| write-pickle | 24 | 13 | 13 | 11 | 4 | 4 |
| k-tree | 29 | 17 | 17 | 15 | 10 | 10 |
| slisp | 45 | 33 | 33 | 23 | 16 | 16 |
| dom | 39 | 25 | 25 | 9 | 7 | 7 |
| postcard | 39 | 15 | 15 | 6 | 1 | 1 |
| m2tom3 | 41 | 23 | 23 | 3 | 1 | 1 |
| m3cg | 32 | 5 | 5 | 5 | 1 | 1 |
| trestle | 23 | 11 | 11 | 8 | 3 | 3 |

4.4.1 *Static Evaluation of* TBAA. Table VII evaluates the relative importance of the three TBAAs: *TypeDecl*: TBAA using only type declarations; *FieldTypeDecl*: TBAA using *TypeDecl* and field declarations; and *SMFieldTypeRefs*: TBAA using *FieldTypeDecl* and assignment statements. For each of the analyses, the table contains the number of static alias pairs determined by the analysis as a percent of all possible alias pairs. Since each memory reference trivially aliases itself, we exclude these pairs from our calculations. In the absence of an alias analysis, the compiler must assume that all possible alias pairs hold (100%). The *Intraprocedural* columns gives the data for intraprocedural aliases: i.e., both references in an alias pair must be in the same procedure. The *Interprocedural* columns give the data when an alias pair may contain references in different procedures. 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 a type-based alias analysis that considers field declarations (*FieldTypeDecl*) is much more precise than the basic TBAA (*TypeDecl*). We also note that selective type merging offers little added precision. Selective type merging reduces the number of local and global alias pairs for `postcard` and reduces global aliases for `m3cg` however, these improvements are so small that they do not show up in the table. 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.

Table VIII evaluates our alias analyses using another static metric: 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 VII and Table VIII, we see that the reduction in alias pairs caused by considering field declarations in TBAA translates into more optimization opportunities: *FieldTypeDecl* finds more redundant loads than *TypeDecl*. The improved precision of selective merges (i.e., *SMFieldTypeRefs*) does not significantly decrease the number of alias pairs and thus does not significantly increase the number of redundant loads removed.

4.4.2 *Dynamic evaluation of* TBAA. This section measures simulated execution-time impact of TBAA on RLE. We measured execution times using a detailed (and validated [Calder et al. 1995]) 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

Table VIII. 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 |
| slisp | 36 | 37 | 37 |
| dom | 328 | 423 | 423 |
| postcard | 259 | 330 | 330 |
| m2tom3 | 369 | 396 | 396 |
| m3cg | 524 | 613 | 613 |
| trestle | 528 | 591 | 591 |

mapped cache. Also, we measured only 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 9 illustrates the simulated execution time impact of TBAA on RLE relative to the original execution time for non-interactive benchmarks. The graph has three bars for each 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). Note that benchmark size increases from left to right on the graph.

TBAA enables RLE to improve program performance from 1% to 8%, and on average 3.6%. Note that the three largest benchmarks benefit the most from the optimizations. 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 VII and Table VIII 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.

4.4.3 *Limit Evaluation of* TBAA. *How much precision does* TBAA *lose in order to achieve its fast time bound?* The speedups for RLE are not impressive and it is easy to contrive examples where TBAA fails to disambiguate memory references while many other alias analyses succeed.

Figure 10 compares heap loads that are redundant at run time *before* and *after* applying RLE. A redundant load occurs when two consecutive loads of the same address load the same value in the same procedure activation. We measure these loads using *ATOM*[Srivastava and Eustace 1994], 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. (We describe this process in more detail elsewhere [Diwan 1996].) In Figure 10, the bars labelled "Redundant originally" give the fraction of heap references (heap loads) that are redundant in the original program and the

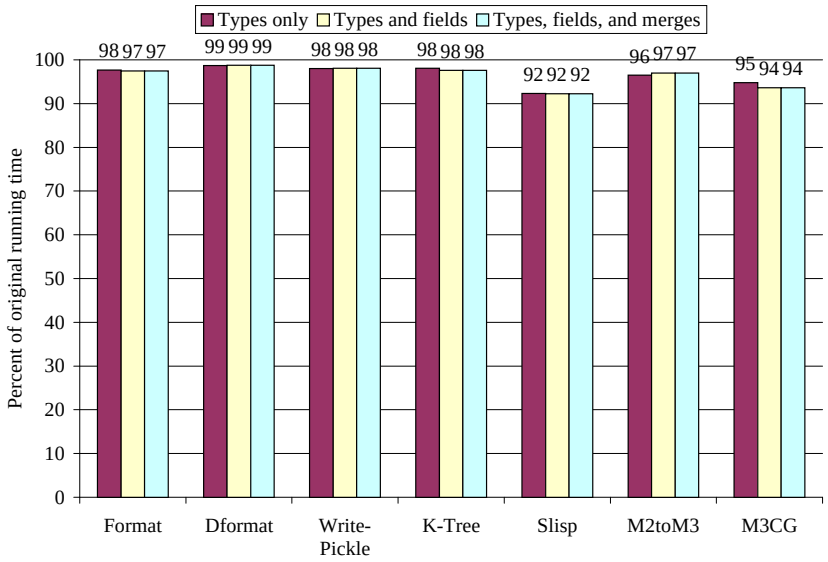


Fig. 9. Impact of RLE

bars labelled "Redundant after optimizations" give the fraction of heap references that are redundant after *SMFieldTypeRefs* and RLE (this fraction is with respect to the original number of heap references). The number above each bar gives the height of that bar. These results are specific to program inputs.

Figure 10 shows that our optimizations eliminate between 35% and 88% 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) **Hidden loads**: RLE could not eliminate a redundant expression because it was implicit in our high-level (AST) intermediate representation. For example, the subscript expression for a Modula-3 open array involves an implicit memory reference to the dope vector.
(2) **No PRE**: 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) **No copy propagation**: RLE did not eliminate a redundant expression because it consisted of multiple smaller expressions and our optimizer does not do copy propagation (recall that RLE eliminates *textually identical* expressions).
(4) **Alias failure**: TBAA did not disambiguate two memory references.

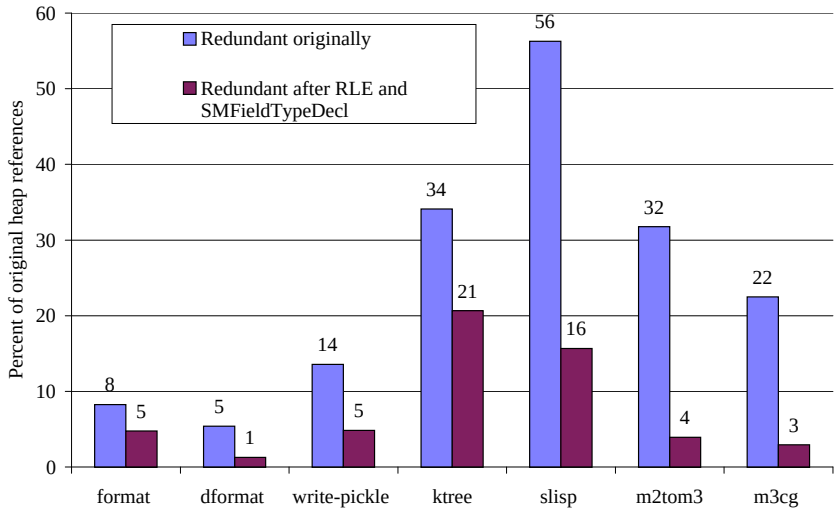


Fig. 10. Comparing TBAA to an Upper Bound

(5) **Rest**: we do not 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 (it is labor intensive).

The first category results from 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 11 breaks down the *Redundant after Optimizations* bar from Figure 10 into the above five categories (the "alias failure" segment was empty in all of our bars and thus we do not even include it in the legend).

Figure 11 illustrates that *Hidden loads* (dope vector accesses to index open arrays) is the most significant source of the remaining redundant loads. Figure 11 also shows that we did not encounter a single situation when optimization failed because of inadequacies in our alias analysis. Those redundant loads that could be because of failed analysis are included in *Rest*, and on average, are less than 2.5% of the remaining loads. Thus, for RLE on these programs and their inputs, there is little room for improvement in our simple and fast alias analysis.

4.4.4 *Summary of Results.* This section evaluated TBAA using four different metrics:

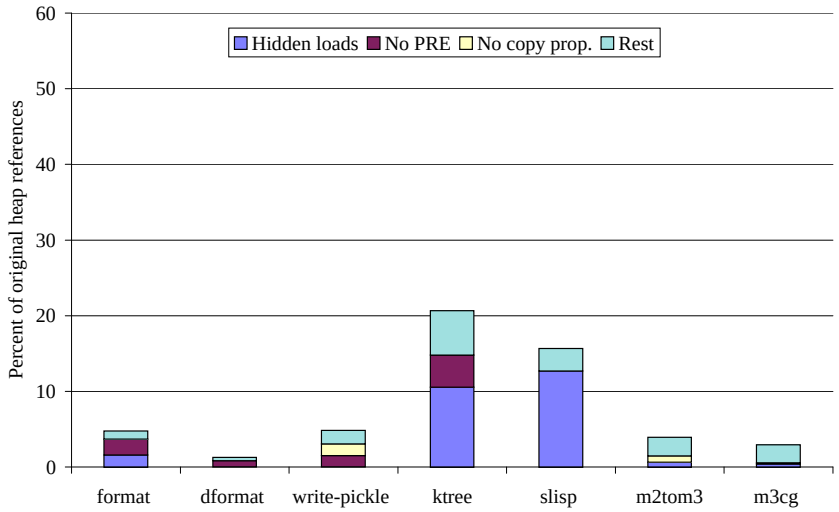


Fig. 11. Source of Redundant Loads after Optimizations

(1) Static alias pairs.
(2) Number of opportunities exposed by TBAA for RLE.
(3) Simulated execution-time improvement resulting from an optimization that uses TBAA (RLE).
(4) An upper-bound for TBAA with respect to RLE.

Each of these four metrics exposes different information about TBAA. The first metric, *static alias pairs*, tells us two things. (1) For our benchmark programs, *SMFieldTypeRefs* 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, *number of opportunities exposed by* TBAA *for* RLE, reveals that *FieldTypeDecl* enables many more opportunities for RLE than *TypeDecl*.

The third metric, *execution-time improvement*, indicates how much an optimization or analysis really matters to the bottom line: performance. Our experiments find that the majority of the execution-time improvement comes from *TypeDecl*. *FieldTypeDecl* improves performance only slightly. The results also illustrate that the execution-time improvement resulting from TBAA and RLE is relatively small:

on average 3.6% improvement.

If we had used only execution-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* would improve performance significantly more than *TypeDecl*.

If we had used only the execution-time improvement results, 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 benchmark programs.

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 any compiler analysis for that matter.

### 4.5 Results for Method Invocation Resolution

Section 4.5.1 uses static and dynamic evaluation to evaluate the effectiveness of our analysis for method resolution. Section 4.5.2 uses limit evaluation to evaluate the analyses. The bar graphs in this section combine dynamic numbers which are represented by the bars with the corresponding static numbers above each bar.

4.5.1 *Static and Dynamic Evaluation.* Figures 12 through 21 illustrate the percent of method invocations resolved by each analysis for each of the benchmark programs. The graphs have one bar for each level of analysis:

| | |
|---|---|
| tha | type hierarchy analysis |
| tha+tpa | **tha** plus intraprocedural type propagation |
| tha+tpa+tbaa | **tha** plus intraprocedural type propagation using TBAA |
| tha+tpa+ip | **tha** plus interprocedural type propagation |
| tha+tpa+ip+tbaa | **tha** plus interprocedural type propagation using TBAA |

The *With NIL* regions in the bars corresponds to the percentage of method invocations at run time that analysis resolves to exactly one procedure. The *Ignoring NIL* corresponds to method invocations that analysis resolves to one user procedure or `error`. The pair above the bar is the number of static call sites (With NIL, Ignoring NIL). The pair includes all method invocation sites including ones that are not executed in our run (and thus not represented in the bars).

The figures illustrate that type-hierarchy analysis resolves many method invocations for most of the benchmark programs. In addition to this, the other analyses benefit different benchmarks (though the benefit is not always visible in the dynamic number but in the static pairs). Intraprocedural type propagation resolves very few additional method invocations (over type hierarchy analysis) but removes many `NIL` possibilities. Thus, type propagation is useful for languages that have well defined semantics for the `NIL` case (such as Modula-3 and Java) but is less useful for other languages (such as C++).

TBAA along with type propagation resolves method invocations in several of the benchmark programs (*dom*, *postcard*, *m3cg*, and *trestle*) though its benefit is visible only in the dynamic numbers for *dom* and *m3cg*. Other runs may display more

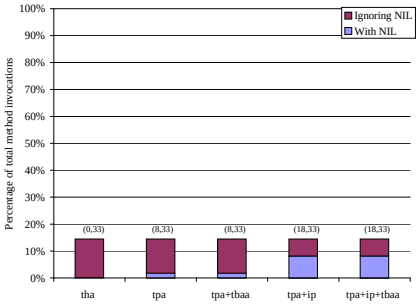


Fig. 12. `format`: Resolved method invocations

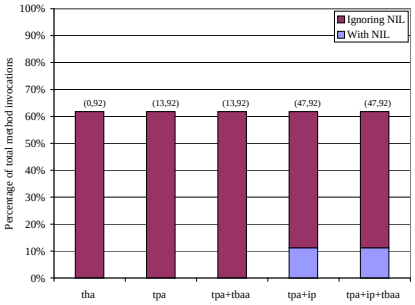


Fig. 13. `dformat`: Resolved method invocations

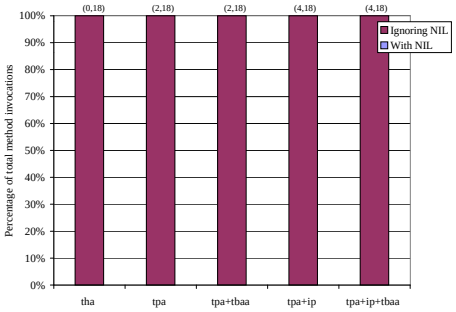


Fig. 14. `write pickle`: Resolved method invocations

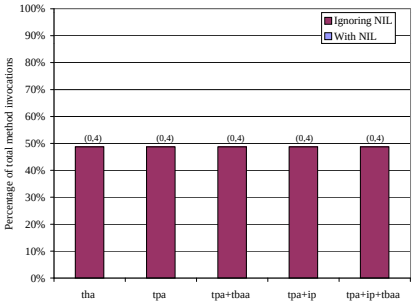


Fig. 15. `ktree`: Resolved method invocations

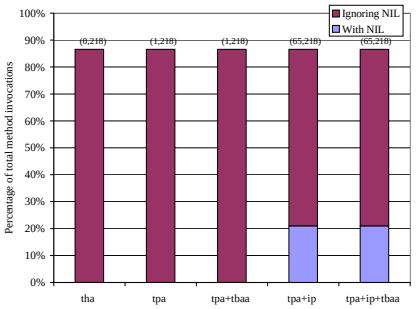


Fig. 16. `slisp`: Resolved method invocations

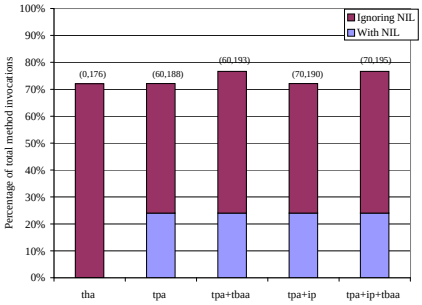


Fig. 17. `dom`: Resolved method invocations

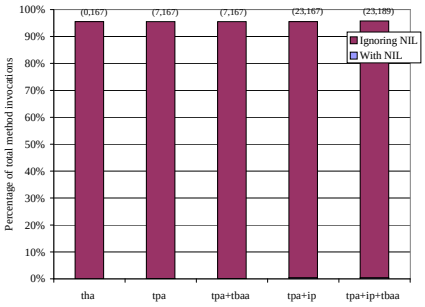


Fig. 18. `postcard`: Resolved method invocations

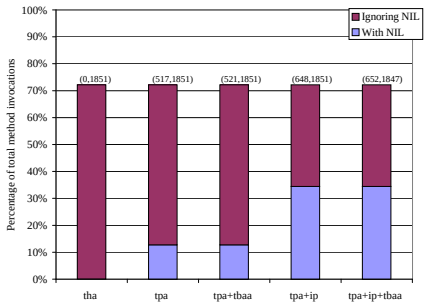


Fig. 19. `m2tom3`: Resolved method invocations

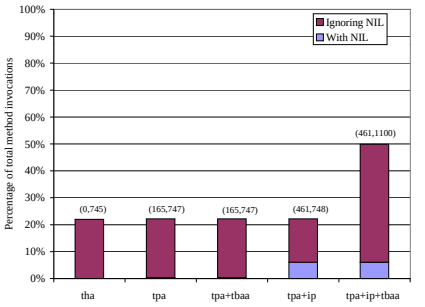


Fig. 20. `m3cg`: Resolved method invocations

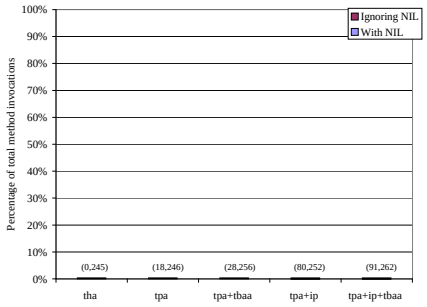


Fig. 21. `trestle`: Resolved method invocations

benefit from TBAA. Interprocedural type propagation also eliminates the NIL possibility in several of the benchmarks, and resolves additional method invocations in *dom*, *postcard*, *m3cg*, and *trestle*. Note that in *dom*, *postcard*, and *m3cg* interprocedural type propagation using TBAA resolves many more method invocations than intraprocedural type propagation using TBAA.

To judge the execution-time impact of the analyses, we ran our non-interactive benchmarks[6] before and after resolution of method invocations on an Alpha 21064 simulator (see Section 4.4.2). In the first experiment, the compiler replaced method invocations that resolved to exactly one user procedure with direct calls. These are the method invocations that make up the *With NIL* region in Figures 12 through 21. The compiler did not convert method invocations that resolved to one user procedure or `error` since that would be inconsistent with Modula-3 language semantics. We found that the execution time improvement averaged less than 2% for the benchmarks even when the compiler inlined the frequently executed resolved method invocations.

In the second experiment, the compiler replaced method invocations that resolved to one user procedure or `error` with direct calls. Ignoring the `error` possibility is inconsistent with Modula-3 semantics but it facilitates comparison with languages such as C++. We found that resolving the method invocations improved performance by 0 to 11%, with an average (arithmetic mean) improvement of 4.6%.

These results show that unlike dynamically-typed languages, the direct cost of method invocations in statically typed-languages is small. The main cost of method invocations is indirect: method invocations obscure control flow and thus inhibit compiler optimizations.

4.5.2 *Limit Evaluation for Method Resolution.* In the absence of control and data merges, such as calls, analysis could determine the allocated type of every variable. However, real programs introduce potential polymorphism by merging control and data as follows:

—Control merges:
  —after a conditional statement
  —at a call site with multiple targets (because of the returns)
  —at a procedure with multiple callers
  —at the return of a procedure with multiple return statements
—Data merges:
  —at assignments through potential aliases (includes heap allocated data, pointers, and array references)

If a merge results in the loss of type information and the affected variable is later used to invoke a method, then that merge is the reason analysis failed to resolve the method invocation. The method invocation may actually be polymorphic, or the analysis may not be powerful enough to resolve it. For every method invocation that our analyses do not resolve, our cause assignment algorithm finds the merges that result in the loss of type information for the receiver of the method invocation. The analyzer finds the merge by following *use-def* chains [Aho et al. 1986] to the point where information is lost.

[6]Because `Trestle`, `postcard`, and `dom` are interactive, we did not include them in this experiment.

| Source | Solution |
|---|---|
| Data merge | More powerful alias analysis |
| Control merge | Context sensitive analysis |
| Unavailable | Analyze libraries |

Table IX. Cause of information loss

We use this information to expose the reason when our analyses fail. The reason suggests which analyses or transformations may be effective on the unresolved method invocations. For example, if a control merge obscures a type, a context sensitive analysis may prevent this loss of information. The cause analysis identifies four sources of information loss:

*Data Merge.* Merge of types due to possible imprecision in analyzing aggregate data structures (recall that the analyses propagate types only to scalars and to some extent to object and record fields),

*Control Merge.* a merge of types resulting from a control merge,

*Code Unavailable.* a conservative type assumed because of unavailability of library code.

Table IX suggests techniques that may prevent the loss of information for each of the three causes of information loss.

Now we address the following questions:

(1) How does our analysis compare to a perfect analysis that resolves *all* monomorphic method invocations?
(2) What transformations could convert the remaining polymorphic method invocations to direct calls?

Figure 22 addresses the first question. Each bar gives the run-time data for one benchmark program. The height of a bar corresponds to the percentage of method invocations that always call the same procedure in a run of the benchmark. Each bar has two regions, the "Resolved" region corresponds to the method invocations resolved by analysis and the "Unresolved" region corresponds to the unresolved monomorphic method invocations. The pair above each bar gives the number of static method invocations corresponding to the two regions. Note that the numbers above the bar only include those method invocations that are executed in our runs. The "Unresolved" region is an upper bound on the truly monomorphic method invocations (*i.e.*, across all possible runs of the programs) that are unresolved by our analyses, and thus on how much better an oracle could do compared to our analyses. It is an upper bound since method invocations may actually be polymorphic on a different program execution or across executions.

Figure 22 shows that, for all benchmarks except `m3cg` and `trestle`, our analysis resolves the vast majority of monomorphic method invocations; the analyses perform almost as well as the oracle. For `format`, `dformat`, `write-pickle`, `slisp`, `dom` and `m2tom3`, our analyses resolve all monomorphic method invocations. Across all the benchmarks, the oracle would resolve at most 11% more method invocation sites compared to our analyses. For the benchmarks where our analyses are less effective, Figure 23 indicates which analyses may be successful in resolving these method invocations.

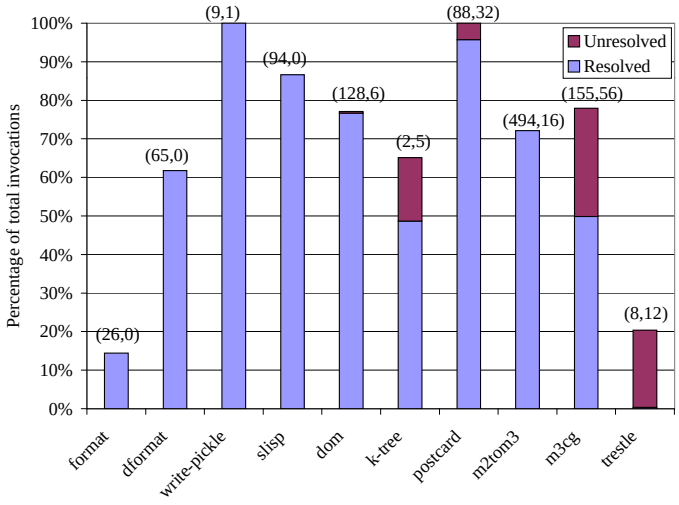


Fig. 22. Monomorphic method invocations

Each bar in Figure 23 breaks down an *unresolved* region in Figure 22 into four regions, one for each cause of analysis failure. The number above each bar is the number of static method invocation sites represented by the bar.

For `m3cg` the figure indicates that a more powerful alias analysis may be successful in resolving more method invocations. On inspection of the source code of `m3cg` we found that an analysis would have to discover the semantics of a stack in order to do better than our alias analysis which is unlikely. Note that like the experiments for RLE, these experiments also suggest that there is little or no room for improvement in TBAA as far as method resolution analyses and our benchmarks are concerned.

For `trestle` and `k-tree`, the primary cause of analysis failure is control merges. and thus a context sensitive analysis may be effective in resolving more method invocations.

Figure 24 addresses the second question: what transformations will be effective in converting the polymorphic method invocations to direct calls? Figure 24 presents data for the method invocation sites that call more than one procedure in a run of the benchmark and thus cannot be resolved by analysis alone. These method invocations are a lower bound on the polymorphic method invocations since in another run of the benchmark, additional method invocations may be polymorphic, although relative execution frequencies may also change. The number above each bar is the number of static method invocation sites corresponding to the method invocations represented by the bar.

Figure 24 illustrates that most run-time polymorphic method invocations arise because more than one type of object is stored in a heap slot. Two techniques,

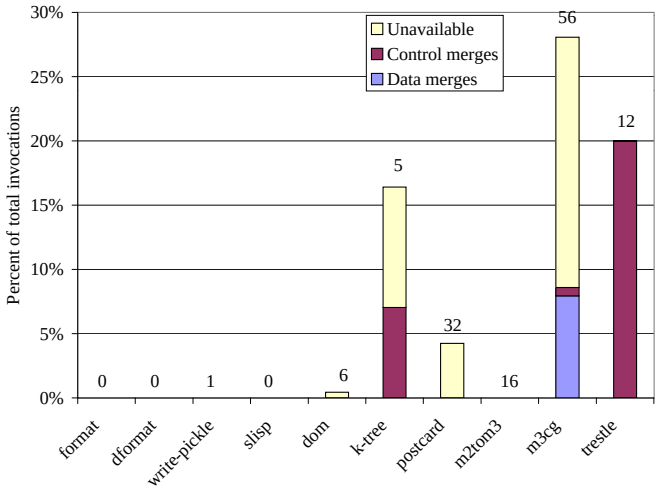


Fig. 23. Monomorphic method invocations that are unresolved

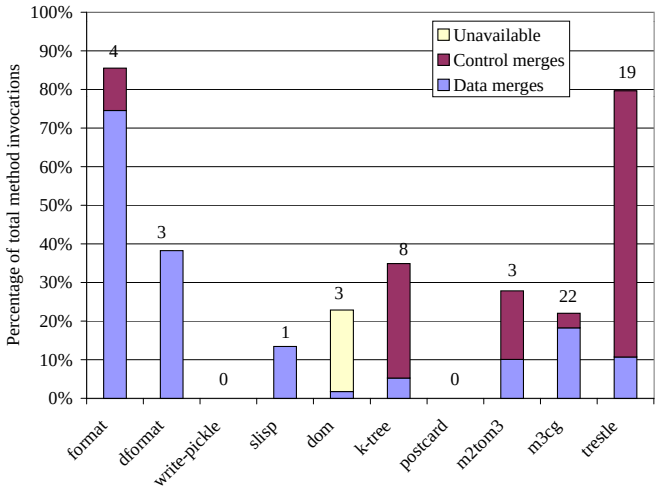


Fig. 24. Polymorphic method invocations

explicit type test [Calder and Grunwald 1994; Hölzle and Ungar 1994] and cloning or splitting combined with aggressive alias analysis, may be able to resolve these method invocations. Merges in control are another important cause of the run-time polymorphism, especially for `trestle`, and can be resolved by code splitting and cloning [Chambers and Ungar 1989; Hall 1991; Chambers and Ungar ].

While the static number of run-time polymorphic sites in the benchmarks is usually small, they are executed relatively frequently. For example, of the 30 method invocation sites executed in a run of `format`, only 4 sites are polymorphic, but they comprise more than 80% of the total method invocations executed. Across all the benchmarks, polymorphic sites are called 26 times more than monomorphic sites. Thus these Modula-3 programs have relatively few polymorphic method invocation sites, but they are executed very frequently. This observation has implication for optimizations: the number of method invocation sites where transformation is needed is small and thus the code growth induced by transformations such as cloning is likely to be negligible.

## 5. CUMULATIVE RESULTS

Figure 25 shows the individual and cumulative impact of three optimizations: method invocation resolution (*Minv*), RLE, and inlining. Inlining replaces direct calls having a high relative frequency with the body of the callee. Since method invocation resolution converts indirect to direct calls, it exposes opportunities for inlining. The numbers in this graph are for the most aggressive versions of method invocation resolution (*tha+tp+tbaa*) and TBAA (*SMFieldTypeRefs*). Since method invocation resolution creates opportunities for both RLE and inlining, we run it first in our experiments. Then we run inlining since it exposes opportunities for RLE. Finally we run RLE.

From this graph we see that that our optimizations together have a significant impact on the speed up our benchmark programs. In particular, the *Minv+Rle+inlining* bars show that our two sets of optimizations improve program performance by as much as 18% with an arithmetic mean of 8%. On comparing the bars, we see that the benefit of combining inlining with our method invocation resolutation and RLE are synergystic: i.e., the performance improvement is greater than the sum of the improvements from the three individual optimizations.

Table X gives the analysis time for our most aggressive combination of analyses: interprocedural type propagation using *SMFieldTypeRefs* (note the time includes both the alias analysis and method resolution). Since part of *SMFieldTypeRefs* happens on demand when a client requests alias information, we cannot easily separate the *SMFieldTypeRefs* time from the method resolution time.

## 6. ANALYZING INCOMPLETE PROGRAMS

In this section, we describe modifications to our alias analysis and method invocation resolution to produce conservative analyses when the entire program is not available. We then show how these changes affect the accuracy of the analyses.

### 6.1 Alias Analysis for Incomplete Programs

All 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

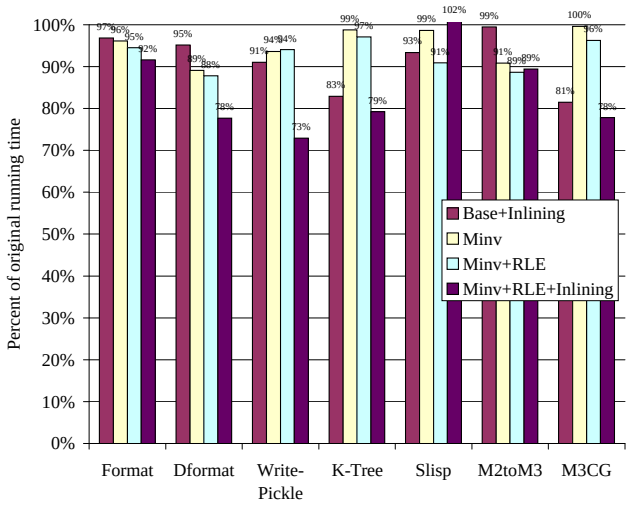


Fig. 25. Cumulative Impact of Optimizations

*closed world assumption.* Many situations arise however in which the entire program is not available: for instance, during separate compilation, or compiling libraries without all their potential clients, or compiling 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

Table X. Analysis time in seconds for interprocedural type propagation and *SM-FieldTypeRefs*

| **Name** | Time (seconds) |
|---|---|
| format | 0.2 |
| dformat | 0.5 |
| write-pickle | 0.3 |
| k-tree | 1.1 |
| slisp | 3.1 |
| dom | 8.9 |
| postcard | 10.2 |
| m2tom3 | 32.7 |
| m3cg | 58.4 |
| trestle | 43.2 |

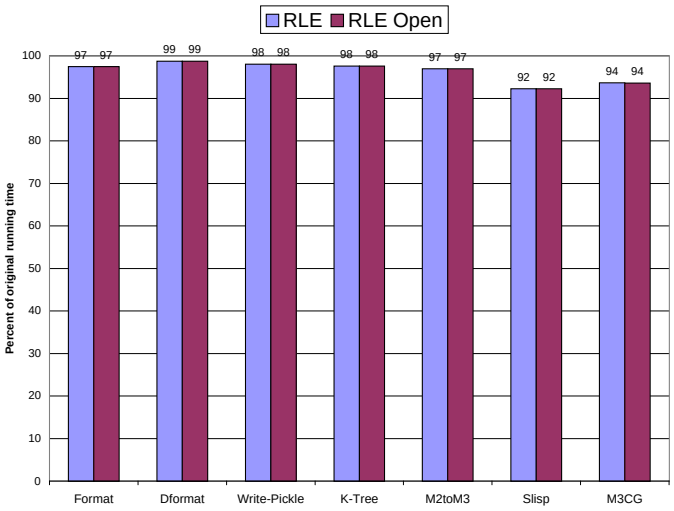


Fig. 26. Simulated execution time using open and closed world assumptions

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 to available code the address of a qualified expression or subscript expression 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 26 compares the simulated run-time improvement resulting from 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 of Table VIII, 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.

### 6.2 Resolving Method Invocations in Incomplete Programs

If the entire program is not available for analysis (in particular, if some of the assignments and type hierarchy are potentially missing), only intraprocedural type propagation (along with the open-world version of TBAA) is applicable. Type propagation must start with the assumption that on entry to each procedure all non-local variables and aggregate locations may have a type that type propagation knows nothing about. However, given the assignments and conditional statements within the procedure, intraprocedural type propagation may still be able to resolve some method invocations.

Figure 27 compares the percent of dynamic method invocations out of all method invocations that our analysis was able to resolve assuming the entire program is available (closed world) and assuming some portion is unavailable (open world). The open world assumption drammatically limits the number of method calls that our analysis can resolve.

## 7. APPLICABILITY TO OTHER LANGUAGES

The analyses described here are language independent but their usefulness depends on the language and the programming style. TBAA, of course, depends greatly on type safety in programs. Thus it is unlikely to be useful for arbitrary C or C++ code. However, if the C++ code is written in a type-safe style, TBAA can be applied to it. To our knowledge, at least two groups of people have applied our ideas to languages other than Modula-3 and found them to be effective: DEC [Reinig 1998] in their C++ compiler and Hosking et al. [Nystrom et al. 1999] in their Java optimizer.

The effectiveness of our method resoultion analyses also depends on programming style and type safety. For example, some C++ programming styles discour-

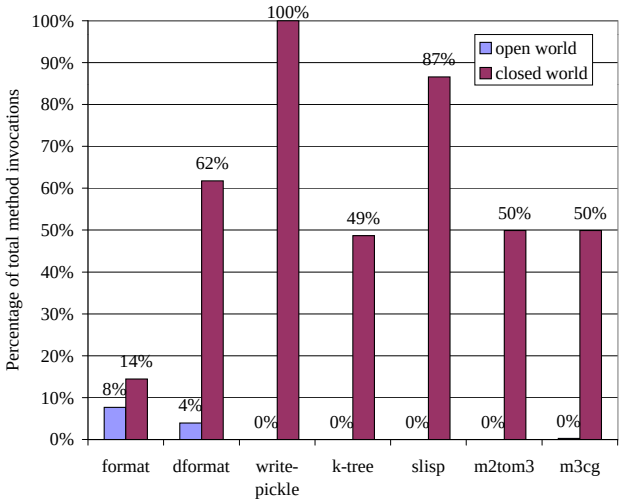


Fig. 27. Percent of resolved method invocations

age the use of virtual functions unless necessary;[7] in essence the style encourages the programmer to attempt type-hierarchy analysis manually. In such situations, the impact of method resolution analyses will be limited compared to Modula-3 programs, where all methods are virtual. We expect that our results will carry over to other statically typed object oriented languages such as C++ *if* the programs are written using only virtual methods. However the execution-time improvement due to our analyses in C++ programs may be greater since method invocations are more costly in languages that have multiple inheritance. Since dynamically-typed languages encourage a fundamentally different style of programming, we expect that our results will not directly apply to them.

## 8. RELATED WORK

In this section, we distinguish our work from others that address alias analysis, method resolution, and compiler optimization evaluation.

### 8.1 Alias Analysis

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 [Banning 1979; Burke et al. 1994; Chatterjee et al. 1999; Chase et al. 1990; Choi et al. 1993; Cooper and

[7] Only virtual functions may be overridden in subtypes.

Kennedy 1989; Deutsch 1994; Emami et al. 1994; Landi and Ryder 1991; 1992; Larus and Hilfinger 1988; Shapiro and Horwitz 1997b; Steensgaard 1996; Weihl 1980][Cooper and Kennedy 1989; Hummel et al. 1994; Cooper and Lu 1997; Larus and Hilfinger 1988; Wilson and Lam 1995]. 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 [Banning 1979; Burke et al. 1994; Chatterjee et al. 1999; Chase et al. 1990; Choi et al. 1993; Cooper and Kennedy 1989; Deutsch 1994; Emami et al. 1994; Landi and Ryder 1991; 1992; Larus and Hilfinger 1988; Shapiro and Horwitz 1997b; Steensgaard 1996; Weihl 1980]. Our upper bound measurement is similar to Wall's [Wall 1991], which assumes a "perfect alias analysis" to find an upper bound on instruction level parallelism. Wall [Wall 1991] 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. [Aho et al. 1986] and Chase et al. [Chase et al. 1990] were among the first to notice that using programming language types could improve alias analysis, but did not present algorithms that did so and did not evaluate it. Our alias algorithm is most similar to those of Rinard and Diniz [Rinard and Diniz 1996], Steensgaard [Steensgaard 1996], and Ruf [Ruf 1995; 1997].

Rinard and Diniz [Rinard and Diniz 1996] 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 [Steensgaard 1996] 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 [Ruf 1997]. Ruf uses his scheme to partition programs for alias analyses, but does not use programming language types in the analysis. Ruf [Ruf 1995] compares a context sensitive alias analysis to a context insensitive alias analysis and finds, for his benchmarks, that they are comparable in precision. We also find that a simple alias analysis can yield very precise results.

Cooper and Lu [Cooper and Lu 1997] 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.

Debray et al. [Debray et al. 1998] 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.

Shapiro and Horwitz [Shapiro and Horwitz 1997a] 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 [Ghiya and Hendren 1998] 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.

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 [Steensgaard 1996].

Since the first publication of some of these algorithms, two groups have applied our techniques to other languages. Reinig [Reinig 1998] describes how to use TBAA in the DEC GEM C and C++ compilers. Reinig applies and uses TBAA intraprocedurally and assumes that the code is compliant with the ANSI standard (TBAA may be turned off if the code violates the ANSI standard). Reinig shows that TBAA combined with other optimizations in GEM yield small improvements in the generated code at an insignificant cost. We think that one of the reasons that they observe less benefit than we do is because the type system in our language (Modula-3) is much richer than the type system in the language of Reinig's experiments (C) and thus we have better information than even type-safe C programs.

Lucassen and Gifford [Lucassen and Gifford 1988] use a type-based analysis to discover expression scheduling constraints. One key difference between our work and theirs is our focus on experimental evaluation of type-based analyses.

Nystrom et al. [Nystrom et al. 1999] apply TBAA to Java programs and use it to do intraprocedural partial redundancy elimination of memory references. Partial redundancy elimination of memory references is more powerful than RLE in that it can eliminate not just fully redundant memory references but also partially redundant memory references. They get modest improvements with TBAA and partial redundancy elimination.

### 8.2 Method Invocation Resolution

Fernandez [Fernandez 1995] and Dean et al. [Dean et al. 1995] evaluate *type hierarchy analysis* for Modula-3 and Cecil respectively. They find that type hierarchy analysis is a worthwhile technique that resolves many method invocations. Our work confirms these results. In addition to type hierarchy analysis, we evaluate a range of other techniques.

Chambers et al. [Chambers et al. 1996] describe and evaluate a range of transformations and analyses for resolving method invocations in object-oriented languages. This paper combines many of the ideas in other papers discussed in this related works section. This paper also serves as an excellent overview of the area. They do not evaluate these algorithms using a limit study.

Palsberg and Schwartzbach [Palsberg and Schwartzbach 1991], Agesen and Hölzle [Agesen and Hölzle 1995], and Plevyak and Chien [Plevyak and Chien 1994] de-

scribe *type inference*[8] for dynamically typed object-oriented languages. Agesen and Hölzle's, and Plevyak and Chien's, analyses are more powerful than ours since they are context sensitive (polyvariant). They are also more complex and expensive. Polyvariant analyses can be used in conjunction with transformations to resolve polymorphic method invocations. Chambers [Chambers 1992], Calder and Grunwald [Calder and Grunwald 1994], Hölzle and Ungar [Hölzle and Ungar 1994], Dean et al. [Dean et al. 1994], and Grove et al. [Grove et al. 1995] describe transformations for converting method invocations to direct calls. We focus solely on analysis here. Plevyak and Chien discuss reasons for loss of type information, but do not present any results. We present detailed data giving reasons for loss of type information.

In work done concurrently with ours, Bacon and Sweeney [Bacon and Sweeney 1996] and Aigner and Hölzle [Aigner and Hölzle 1996] evaluate techniques for resolving method invocations in C++ programs. Bacon and Sweeney evaluate three fast analyses, including type hierarchy analysis, for resolving method invocations in C++ programs. Unlike us, Bacon and Sweeney evaluate only flow insensitive analyses. Aigner and Hölzle evaluate type feedback and type hierarchy analysis and find that they are both effective at resolving method invocations.

Driesen and Hölzle [Driesen and Hölzle 1996] report on the direct cost of virtual function calls in C++ programs. They find that in "all virtual" versions of programs, the median direct overhead of virtual functions is 13.7%. These numbers are somewhat higher than what we observe for Modula-3 programs, and may be caused by C++'s multiple inheritance which makes virtual function calls more expensive. Of course, speedup numbers are also highly dependent on the benchmark suite.

Shivers [Shivers 1991] describes and classifies a range of analyses to discover control flow in Scheme programs. Our interprocedural type propagation is similar to his 0CFA. While Shivers focuses on powerful (and slow) analyses—0CFA is the least powerful analysis he considers—we focus on simple and fast analyses. Interprocedural type propagation is the most complicated analysis we consider.

Pande and Ryder's algorithm [Pande and Ryder 1995] performs pointer analysis at the same time as method invocation analysis. Plevyak and Chien's type inference algorithm also does some pointer analysis [Plevyak and Chien 1994]. Both algorithms consider the control flow in a program and are thus more powerful than TBAA but are also slower. On a SPARC-10, Pande and Ryder's algorithm can take 23 minutes to analyze programs that are less than 1000 lines of code (median 36 seconds). Our most aggressive analysis takes 43 seconds to analyze 28,977 lines of code on a DEC 3000/400. In subsequent work [Chatterjee and Ryder 1997a; 1997b; Chatterjee et al. 1999] Ryder's group has worked on improving the scalability of their analyses. We show that our simple analysis is effective and there is little to be gained by a more powerful analysis for our benchmarks. This result originates in part from Modula-3's language semantics, which restrict aliasing; a more powerful alias analysis may be more useful for C++ than for Modula-3, but this need has not yet been demonstrated for significant applications.

DeFouw et al. [DeFouw et al. 1998] describes a parametrized framework that

[8] "Type propagation" and "type inference" are terms that have been used to describe the same kinds of analysis in object-oriented languages.

integrates a range of analyses for method resolution. They use this framework to evaluate a range of analyses and find that there is little difference in bottom line performance impact between their analyses. Our results support theirs: for many applications a fast and simple alias analysis may be sufficient.

A final key difference between our work and that of others is that we present results that give the reason when analysis fails, and place upper bounds on how well more powerful analyses or transformations can possibly do.

### 8.3 Evaluating Optimizations

Larus and Chandra [Larus and Chandra 1993] introduce a technique, compiler auditing, that uses studies to test compiler optimizations. This technique is very similar to our limit studies and in particular their method of auditing redundant loads and stores is similar to the oracle we use to evaluate TBAA and RLE. One difference is that Larus and Chandra are pessimistic about procedure calls whereas we are optimistic.

## 9. CONCLUSIONS

We described and evaluated 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, *SMFieldTypeRefs*, 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. TBAA with redundant load elimination (RLE), produces modest performance improvements, but TBAA is precise for our benchmarks; 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.

We also described and evaluated a range of analyses for resolving method invocations: *type-hierarchy analysis*, and *intraprocedural and interprocedural type propagation* with and without TBAA. On average, our analyses resolve more than 92% of the method invocation sites that are amenable to analysis and improve the runtime of the benchmark programs by up to 19%. For method invocations that are unresolved by our analyses, we determine the reason for analysis failure. The failure reason suggests which other analyses and transformations may be effective. The primary failure reason in our benchmarks is polymorphism: the method invocations called more than one procedure at run time and thus are not amenable to analysis alone. Most of this polymorphism is due to objects of different types being stored in heap slots. The other significant reasons for analysis failure are an insufficiently powerful pointer analysis and lack of a context sensitive analysis. Improving the pointer analysis and adding a context sensitive analysis would resolve at most 7% more method invocation sites. Our results show that the method resolution analyses are significantly less effective with an open-world assumption instead of a closed world assumption.

In summary, we have shown that simple, fast type-based analyses are an effective tool for optimizing object-oriented programs.

## 10. ACKNOWLEDGMENTS

We would like to thank Ole Agesen and Darko Stefanovic for comments on drafts of this paper.

## REFERENCES

AGESEN, O. 1995. Concrete type inference: Delivering object-oriented applications. Ph.D. thesis, Stanford University, Palo Alto, CA.

AGESEN, O. AND HÖLZLE, U. 1995. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, Austin, Texas, 91–107.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

AIGNER, G. AND HÖLZLE, U. 1996. Eliminating virtual function calls in C++ programs. In *Proceedings of European Conference on Object-Oriented Programming*. Linz, Austria.

BACON, D. AND SWEENEY, P. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM SIGPLAN '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, ACM Press, San Jose, CA.

BANNING, J. 1979. 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*. San Antonio, Texas, 29–41.

BATES, R. M. 1994. K-trees. Personal communication.

BURKE, M., CARINI, P., CHOI, J.-D., AND HIND, M. 1995. Interprocedural pointer alias analysis. Tech. rep., IBM T.J. Watson Research Center, Yorktown Heights, NY. Aug.

BURKE, M., CARINI, P. R., CHOI, J.-D., AND HIND, M. 1994. Efficient flow-insensitive alias analysis in the presence of pointers. Tech. Rep. 19546, IBM T.J. Watson Research Center, Yorktown Heights, NY. Sept.

CALDER, B. AND GRUNWALD, D. 1994. Reducing indirect function call overhead in C++ programs. In *21st Symposium on Principles of Programming Languages*. ACM, Portland, Oregon, 397–408.

CALDER, B., GRUNWALD, D., AND EMER, J. 1995. A system level perspective on branch architecture performance. In *28th International Symposium on Microarchitecture*. 199–206.

CARINI, P. R., SRINIVASAN, H., AND HIND, M. 1995. Flow-sensitive type analysis for C++. Tech. Rep. RC 20267, IBM T.J. Watson Research Center, Yorktown Heights, NY. Sept.

CHAMBERS, C. 1992. The design and evaluation of the SELF compiler, an optimizing compiler for object-oriented programming languages. Ph.D. thesis, Stanford University, CA.

CHAMBERS, C., DEAN, J., AND GROVE, D. 1996. Whole-program optimization of object-oriented languages. Tech. Rep. 96-06-02, University of Washington, Seattle, Washington. June.

CHAMBERS, C. AND UNGAR, D. Making pure object oriented languages practical. In *Proceedings of the ACM SIGPLAN '91 Conference on Object-Oriented Programming Systems, Languages, and Applications*. 1–15.

CHAMBERS, C. AND UNGAR, D. 1989. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*. 146–160.

CHAMBERS, C. AND UNGAR, D. 1990. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*. 150–164.

CHASE, D. R., WEGMAN, M., AND ZADECK, F. K. 1990. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*. 296–310.

CHATTERJEE, R., RYDER, B. G., AND LANDI, W. A. 1999. Relevant context inference. In *Proceedings of 26th ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*. ACM.

CHATTERJEE, R. K. AND RYDER, B. G. 1997a. Modular concrete type-inference for statically typed object-oriented programming languages. Tech. Rep. DCS-TR-349, Rutgers University. Nov.

CHATTERJEE, R. K. AND RYDER, B. G. 1997b. Scalable, flow-sensitive type-inference for statically typed object-oriented programming languages. Tech. Rep. DCS-TR-326, Rutgers University. July.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. 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*. Charleston, SC, 232–245.

COOPER, K. AND LU, J. 1997. Register promotion in c programs. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*. Las Vegas, Nevada.

COOPER, K. D. AND KENNEDY, K. 1989. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*. 49–59.

DEAN, J., CHAMBERS, C., AND GROVE, D. 1994. Identifying profitable specialization in object-oriented languages. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. Association of Computing Machinery, Orlando, FL.

DEAN, J., DEFOUW, G., GROVE, D., LITVINOV, V., AND CHAMBERS, C. 1996. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the ACM SIGPLAN '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*. San Jose, CA, 83–100.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of European Conference on Object-Oriented Programming*. Aarhus, Denmark, 77–101.

DEBRAY, S., MUTH, R., AND WEIPPERT, M. 1998. Alias analysis of executable code. In *Conference Record of the Twentyfifth Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*.

DEFOUW, G., GROVE, D., AND CHAMBERS, C. 1998. Fast interprocedural class analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 222–236.

DEUTSCH, A. 1994. Interprocedural May-Alias analysis for pointers: Beyond $k$-limiting. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. 230–241.

DIWAN, A. 1996. Understanding and improving the performance of modern programming languages. Ph.D. thesis, University of Massachusetts, Amherst, MA 01003.

DRIESEN, K. AND HÖLZLE, U. 1996. The direct cost of virtual function calls in c++. In *Proceedings of the ACM SIGPLAN '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*. San Jose, CA.

EMAMI, M., GHIYA, R., AND HENDREN, L. J. 1994. 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*. 242–256.

FERNANDEZ, M. F. 1995. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of Conference on Programming Language Design and Implementation*. SIGPLAN, ACM Press, La Jolla, CA, 103–115.

GHIYA, R. AND HENDREN, L. J. 1998. Putting pointer analysis to work. In *Conference Record of the Twentyfifth Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*.

GROVE, D., DEAN, J., GARRETT, C., AND CHAMBERS, C. 1995. Profile-guided receiver class prediction. In *Proceedings of the ACM SIGPLAN '95 Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, Austin, Texas, 108–123.

HALL, M. W. 1991. Managing interprocedural optimizations. Ph.D. thesis, Rice University, Houston, Texas.

HENNESSY, J. AND PATTERSON, D. 1995. *Computer Architecture A Quantitative Approach.* Morgan-Kaufmann.

HÖLZLE, U. AND UNGAR, D. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.* ACM, 326–336.

HUMMEL, J., HENDREN, L. J., AND NICOLAU, A. 1994. 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.* 218–229.

KAM, J. B. AND ULLMAN, J. D. 1976. Global data flow analysis and iterative algorithms. *Journal of the ACM 7,* 3, 305–318.

LANDI, W. AND RYDER, B. G. 1991. Pointer-induced aliasing: a problem classification. In *Conference Record of the Eighteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages.* Orlando, FL, 93–103.

LANDI, W. AND RYDER, B. G. 1992. Interprocedural side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.* San Francisco, CA, 235–248.

LARUS, J. R. AND CHANDRA, S. 1993. Using tracing and dynamic slicing to tune compilers. University of Wisconsin Technical Report 1174.

LARUS, J. R. AND HILFINGER, P. N. 1988. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.* Atlanta, GA, 21–34.

LISKOV, B. AND GUTTAG, J. 1986. *Abstraction and Specification in Program Development.* MIT Press.

LUCASSEN, J. M. AND GIFFORD, D. 1988. Polymorphic effect systems. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, 47–57.

NAYERI, F., HURWITZ, B., AND MANOLA, F. 1994. Generalizing dispatching in a distributed object system. In *Proceedings of European Conference on Object-Oriented Programming.* Bologna, Italy, 450–473.

NELSON, G., Ed. 1991. *Systems Programming with Modula-3.* Prentice Hall, New Jersey.

NYSTROM, N., HOSKING, A. L., WHITLOCK, D., CUTTS, Q., AND DIWAN, A. 1999. Partial redundancy elimination for access path expressions. In *Proceedings of the International Workshop on Aliasing in Object-Oriented Systems.* Lisbon, Portugal. revision of Purdue University Computer Sciences Technical Report 98-044.

OXHOJ, N., PALSBERG, J., AND SCHWARTZBACH, M. I. 1992. Making type inference practical. In *Proceedings of European Conference on Object-Oriented Programming.* Springer-Verlag, 329–349.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1991. Object-oriented type inference. In *Proceedings of the ACM SIGPLAN '91 Conference on Object-Oriented Programming Systems, Languages, and Applications.* SIGPLAN, ACM Press, Pheonix, Arizona, 146–162.

PANDE, H. AND RYDER, B. G. 1995. Static type determination and aliasing for C++. Tech. Rep. LCSR-TR-250, Rutgers University. July. A version of this appeared in *Proceedings of the Third International Static Analysis Symposium (SAS'96).*

PANDE, H. D. 1996. Compile time analysis of c and c++ systems. Ph.D. thesis, Rutgers, The State University of New Jersey, New Brunswick, NJ.

PLEVYAK, J. AND CHIEN, A. 1994. Precise concrete type inference for object-oriented languages. In *Proceedings of the ACM SIGPLAN '94 Conference on Object-Oriented Programming Systems, Languages, and Applications.* ACM, 324–340.

REINIG, A. G. 1998. Alias analysis in the dec c and digital c++ compilers. *DIGITAL Technical Journal 10,* 1 (Dec.).

RINARD, M. C. AND DINIZ, P. C. 1996. 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.

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

Ruf, E. 1997. Partitioning dataflow analyses using types. In *popl97*. Paris, France.

Shapiro, M. and Horwitz, S. 1997a. The effects of the precision of pointer analysis. In *Lecture Notes in Computer Science, 1302*, P. V. Hentenryck, Ed. Springer-Verlag, 16–34. Proceedings from the *4th International Static Analysis Symposium*.

Shapiro, M. and Horwitz, S. 1997b. 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.

Shivers, O. 1991. Control-Flow Analysis of Higher-Order Languages. Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA.

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

Stallman, R. M. *Gnu C Compiler*. Free Software Foundation, Cambridge, MA. Software distribution.

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

Sun Microsystems Computer Corporation 1995. *The Java language specification*, 1.0 Beta ed. Sun Microsystems Computer Corporation.

Wall, D. W. 1991. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Santa Clara, California, 176–189.

Weihl, W. E. 1980. 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*. Las Vegas, Nevada, 83–94.

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