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. Object-oriented programs make extensive use of method invocations and pointer dereferences, both of which are potentially costly on modern machines. 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 type-based alias analysis to disambiguate memory references. It uses the results of the type-based alias analysis to eliminate redundant memory references and to replace monomorphic method invocation sites with direct calls. Using limit, static, and running time evaluation, we demonstrate that these techniques are effective, and sometimes perfect for a set of Modula-3 benchmarks.

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 make extensive use of pointers, type hierarchies, and virtual method invocations to improve code reuse and correctness.

This work was supported by the National Science Foundation under grants CCR-9211272, CCR-9525767, and ITR CCR-0085792 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 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.
Portions of this paper appeared previously [Diwan et al. 1996; Diwan et al. 1998].
Authors' addresses: A. Diwan, Department of Computer Science, University of Colorado, Boulder, CO 80309; email: diwan@cs.colorado.edu. K. McKinley and E. Moss, Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610; email: {mckinley,moss}@cs.umass.edu.

These features have a cost. For example, without alias analysis, the compiler must assume all pointer dereferences are potential aliases and may not reorder them. Compilers need to reorder instructions to effectively exploit the underlying hardware, which may have multiple issue functional units, and elaborate pipelines. An effective alias analysis disambiguates memory references, and enables the compiler to reorder pointer accesses.

Virtual method invocations are costly as well. Method invocations obscure which procedure is actually being invoked. In programs written in pure object-oriented languages, method look-up is costly in itself because method invocations are frequent [Chambers 1992]. However, in programs written in hybrid object-oriented languages, method invocations are typically less frequent and therefore do not have a significant cost. However, for all object-oriented languages, 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 resulting from pointer dereferences and method invocations, we present a range of *type-based* alias analyses (TBAA). TBAA uses programming-language types. Our alias analysis techniques range from a simple inspection of the type hierarchy to interprocedural flow-insensitive analysis. We determine the effectiveness and usefulness 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. Method resolution replaces monomorphic method invocations with direct calls. To better understand the impact of TBAA on method resolution, we consider three different algorithms for method resolution and extend two of them with TBAA. These method resolution algorithms range from a simple inspection of the type hierarchy to a new interprocedural flow-sensitive context-insensitive analysis. While there are obvious interactions between pointer analysis and method resolution, we pick a fixed order for the analyses: TBAA followed by method resolution analysis. Previous work proposes a few of our alias analyses and method resolution techniques, but our evaluation reveals new insights about these and our new algorithms.

We evaluate the effectiveness of TBAA for RLE and method resolution using static, dynamic, and limit analyses. This evaluation methodology is more thorough than most of the previous work on alias analysis. Our results show that there is surprisingly little room for improvement in TBAA for our benchmarks. For example, a better alias analysis would perform better than TBAA for method resolution in at most three of our 10 benchmark programs. Although others have proposed using types in similar ways, we believe we are the first to demonstrate their unanticipated effectiveness for important optimizations. We also modify our analyses to work on incomplete programs and demonstrate that the effectiveness of TBAA and RLE is not compromised, but that method resolution is not effective on incomplete programs. We have implemented our analyses in a traditional optimizing compiler for Modula-3. The speed and simplicity of these analyses also makes them practical for statically compiled Java programs. For Java programs that use dynamic class loading, our modifications for incomplete programs are applicable, but will

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`

probably be less effective.

The remainder of this paper is organized as follows. Section 2 gives a brief technical background on types. Section 3 describes our type-based alias analysis algorithms. It discusses three progressively more precise alias analyses based on type declarations, field declarations and other high-level properties, and flow-insensitive data-flow analysis. Section 4 describes two uses of TBAA: RLE and method resolution. It also describes algorithms for method resolution, the most aggressive of which use TBAA. Section 5 presents our experimental methodology. Section 6 evaluates TBAA using static, dynamic, and upper bound evaluation for each of RLE, method resolution, and inlining enabled by method resolution. Section 7 evaluates TBAA using our optimizations when the entire program is not available for analysis. Section 8 considers how our techniques apply to other optimizations and object-oriented languages, particularly C++ and Java. Section 9 discusses related work in alias analysis and method resolution. Section 10 concludes.

2. BACKGROUND

We now present some assumptions and terminology that we will use in the rest of the paper. All of our analyses assume the entire program is available unless otherwise stated. Section 2.1 describes what memory references look like in the language that we analyze, Modula-3 [Nelson 1991]. Section 2.2 describes how method invocations give rise to polymorphism in Modula-3 programs.

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] Without loss of generality, we assume that all pointer dereferences are explicit and that a variable declared to be of object or array type actually contains the object or array rather than a pointer to the object or array. Modula-3 has implicit pointer dereferences, but at the intermediate representation level all pointer dereferences are explicit.

We call a nonempty string of memory references, for example `a^.b[i].c`, an *access path* ($\mathcal{AP}$) [Larus and Hilfinger 1988] and assume that object fields have different names. We define:

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

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

```
TYPE U = OBJECT
           f: U;
         METHODS
           m := mU;
         END;
(* V is a subtype of U *)
TYPE V = U OBJECT
         METHODS
           n := nV;
         OVERRIDES
           m := mV;
         END;
```

Fig. 1. A Modula-3 type hierarchy.

In Modula-3 and other type-safe languages, a variable of type `REF T` can legally point to 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 Polymorphism through Subtyping

Statically typed object-oriented languages support polymorphism through subtyping. A variable of type `S` where `S` is a subtype of `T` supports all the behavior of `T` and may extend it. 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 `REF T` may point to objects that are subtypes of `T`.

Consider the Modula-3 type hierarchy in Figure 1, which defines a type `U`, and `V`, a subtype of `U`. `V` has all the behavior of `U` (in particular, the `m` method) but has a different implementation of `m` (`mV` instead of `mU`). `V` also supports the `n` method, which `U` does not. Invoking the `m` method on a variable with declared type `REF U` may invoke one of three procedures:

(1) `mU`, if the variable is currently a pointer to an object of type `U`;

(2) `mV`, if the variable is currently a pointer to an object of type `V`; or

(3) `error`, if variable is currently a pointer of type `NULL`.

In general, invoking a method on a variable of type `REF U` (the *receiver*) can call any procedure that overrides that method in *Subtypes*(`U`). The `NULL` type, which contains a single value `NIL`, is a subtype of all reference types in Modula-3 and overrides all methods with an `error` procedure. While `NULL` is a type in Modula-3 and `NIL` is a value, we will abuse these terms and use `NULL` to mean both the type and the value when it is clear from the context.

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 with different implementations of the `print` method, then the `print` method invokes different procedures depending on the type of the list element.

A *monomorphic* method invocation site always invokes the same user procedure (or `error`), for all possible program executions. The receiver need not always be the same *type*, but the method implementation must be the same. To continue the

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

Fig. 2. Type hierarchy example.

linked list example, if the list links objects of only one type, then the `print` method will always invoke the same procedure.

We will call a method invocation site *run-time monomorphic* if, over some set of program runs, it always invokes the same method implementation. Thus a monomorphic site will always be run-time monomorphic, but a run-time monomorphic site may be polymorphic because it may invoke a different method implementation in some execution not yet considered. Whether a method invocation site is monomorphic is undecidable in general; we will thus find a conservative estimate. A method invocation is *resolved* if it is identified as being monomorphic. Section 4.2 gives algorithms for resolving monomorphic sites.

3. 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 first describe three progressively more powerful versions of TBAA and then conclude with their complexity.

3.1 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 that $\mathcal{AP}$ may reference any object with the same declared type or subtype. This version of TBAA we call T-TBAA. More formally, given two $\mathcal{AP}$s p and q, T-TBAA determines that they are *aliases* if and only if T-TBAA (p, q) evaluates to true:

$$\text{T-TBAA}\ (p,\ q) = \mathit{Subtypes}(\mathit{Type}(p)) \cap \mathit{Subtypes}(\mathit{Type}(q)) \neq \emptyset.$$

Consider the example in Figure 2. Since `S1` is a subtype of `T`, variables of type `REF T` can point to objects of type `REF S1`. Thus,

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

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 T-TBAA is not transitive.

Table II. TF-TBAA ($\mathcal{AP}1$, $\mathcal{AP}2$) Algorithm

Case	$\mathcal{AP}$ 1	$\mathcal{AP}$ 2	TF-TBAA($\mathcal{AP}$ 1, $\mathcal{AP}$ 2)
1	p	p	true
2	p.f	q.g	(f = g) ∧ TF-TBAA (p, q)
3	p.f	q^	*AddressTaken* (p.f) ∧ T-TBAA (p.f, q^)
4	p[i]	q^	*AddressTaken*(p[i]) ∧ T-TBAA (p[i], q^)
5	p.f	q[i]	false
6	p[i]	q[j]	TF-TBAA (p, q)
7	x	y	x = y
8 (otherwise)	p	q	T-TBAA (p, q)

3.2 Using Field Access Types

We next improve the precision of T-TBAA using the type declarations of fields and other high-level information in the program. This version of TBAA we call TF-TBAA. The TF-TBAA algorithm appears in Table II. Given *AP 1* and *AP 2*, it returns true if *AP 1* and *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 that p can possibly refer. *AddressTaken*(q[i]) returns true if the program takes the address of some element of an array that q can possibly refer. In Modula-3, programs may take the addresses of memory locations in only two ways: via the pass-by-reference parameter-passing mechanism, and via the WITH statement, which creates a temporary name for an expression. Note, that unlike T-TBAA, which needs only the type hierarchy, *AddressTaken* actually needs to look at all the instructions in the program. For simplicity, we assume that aggregate accesses, such as assignments between two records, have been broken down into accesses of each component.

The eight 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. Note that this case recursively uses TF-TBAA to more precisely handle the aliasing of access paths such as a.x.g and a.y.g.

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. TF-TBAA ignores the actual subscripts. Note that this case recursively uses TF-TBAA to more precisely handle the aliasing of access paths such as a.x[i] and a.y[j].

7: Two distinct variables are never aliases

8 (otherwise): For all other cases of $\mathcal{AP}$s, including two pointer dereferences, TF-TBAA uses T-TBAA to determine aliases.

The Java programming language will have similar rules though we will need additional mechanisms to handle programs that use dynamic class loading and reflection. For C++ the rules must be more conservative to handle arbitrary pointer casts and pointer arithmetic.

3.3 Using Assignment

T-TBAA is conservative in the sense that it assumes that the program uses types in their full generality. For instance, a program might use a list package capable of linking objects of different types, and in fact link objects of only one type. We thus improve on T-TBAA 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 TM-TBAA. Unlike T-TBAA, which always merges the declared type of an $\mathcal{AP}$ with all of its subtypes, TM-TBAA only merges a type with a subtype when a statement assigns some pointer to subtype `S` to a variable declared to be of type `REF T`. As an example, consider applying T-TBAA to the following program using the type hierarchy in Figure 2:

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

T-TBAA assumes that `t^` and `s^` may refer to the same location. By inspecting the code however, it is obvious that `t` and `s` never point to the same location. TM-TBAA proves independence in this situation as follows: if the program never assigns a value of type `REF S1` to a location of type `REF T` (directly or indirectly), then `t^` and `s^` cannot possibly be aliases. If there is any such assignment, TM-TBAA ignores the control flow and assumes an alias. We call these assignments *merges*.

Figure 3 presents the algorithm to merge types selectively for complete programs.[2] The 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, TM-TBAA determines that they are aliases if and only if TM-TBAA (p,q) evaluates to true:

$$\text{TM-TBAA}\ (p,q) = \mathit{TypeRefsTable}(\mathit{Type}(p)) \cap \mathit{TypeRefsTable}(\mathit{Type}(q)) \neq \emptyset$$

In Figure 3, each set $\mathcal{T} = \{T_1, \ldots, T_k\}$ in *Group* represents an equivalence class of types such that an $\mathcal{AP}$ with a declared type $T \in \mathcal{T}$ may refer to any object of type U such that $U \in \mathcal{T}$. For example, given the set $\mathcal{T} = \{$`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. 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 TM-TBAA relationship.[4]

[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] This step is similar to Steensgaard's algorithm [Steensgaard 1996].

[4] Steensgaard's algorithm [Steensgaard 1996] applied to user-defined types would not discover this asymmetry.

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

(* Step 2: merge sets because of assignments *)
for each implicit and explicit pointer assignment a:=b do
  let Type(a) be REF Ta and Type(b) be REF Tb;
  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 REF T do
  let g ∈ Group, T ∈ g
    TypeRefsTable(T) = g ∩ Subtypes(T)
```

Fig. 3. Selective type merging.

For instance, an $\mathcal{AP}$ with declared type `REF T` in Figure 2 may point to objects of type `T` or type `S1`, but an $\mathcal{AP}$ declared as `REF S1` may not point to objects of type `T`. The final result of Step 3 is the *TypeRefsTable*.

Figure 4 uses the type declarations in Figure 2 to illustrate how the selective merging algorithm works. Step 1 initializes each declared type to be in a set of its own, as shown in Figure 5(a) where each oval represents a set in *Group*. Figure 5(b) shows *Group* after Step 2 merges types `T` and `S1`, the types for the first assignment; and Figure 5(c) shows that the second assignment causes Step 2 to merge `S2` with `T` and `S1`. `S3` remains in a set by itself. Step 3 of the merge algorithm then creates asymmetry for the subtype declarations in the *TypeRefsTable*, as shown in Figure 4. Notice that TM-TBAA determines that $\mathcal{AP}$s declared to point to `T` may not point to objects of type `S3`, but T-TBAA assumes they may.

We obtain the final version of our TBAA algorithm TFM-TBAA by using TM-TBAA instead of T-TBAA in the TF-TBAA algorithm of Table II.

3.4 Complexity of Analyses

The complexity of the slowest TBAA (TFM-TBAA) is dominated by Step 2 of TM-TBAA (Figure 3). 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 $O(n * |T|)$, where n is the number of instructions in the program and $|T|$ is the number of types in the program. If we use a fast union-find data structure [Tarjan 1975] (instead of our current bit vector set implementation) we can further reduce the complexity of this analysis to near-linear time. The time to *use* the results of the TBAA may, of course, be more than near-linear. For instance, computing all the *may-alias* pairs using TBAA, or any other *points-to* analysis, takes $O(e^2)$ steps, where e is the number of memory expressions in the program and each step requires querying the results of the points-to analysis.

4. USING TBAA

Most compiler analyses and optimizations can benefit from alias analysis. In this section, we describe two optimizations, redundant load elimination (RLE) and

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

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

Step 3: *TypeRefsTable*

Fig. 4. Example to illustrate TM-TBAA.

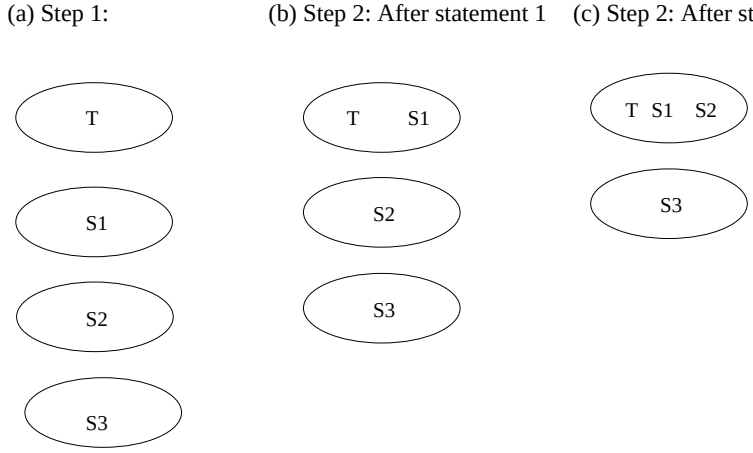

Fig. 5. Selective merging for Figure 4.

method resolution, that use TBAA.

4.1 Redundant Load Elimination

RLE combines variants of loop-invariant code motion and common subexpression elimination [Aho et al. 1986], but applies them to loads instead of 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].

Similar to register promotion [Cooper and Lu 1997], RLE hoists a memory reference out of a loop if it is loop invariant and is executed on every iteration of the loop, leaving it up to the back end to place the hoisted memory reference in a register. For example in Figure 6, the access path `a^.b` is redundant on all paths, and loop-invariant code motion moves it into the loop header. As shown in Figure 7, RLE also eliminates common subexpressions of memory references. 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 back end. To enable RLE across calls, RLE is preceded by a mod-ref analysis that summarizes

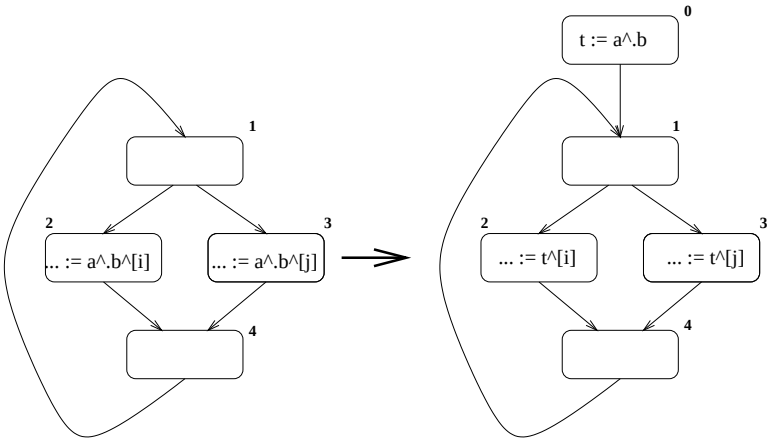

Fig. 6. Eliminating loop-invariant memory loads.

the objects (in terms of types and fields) that are referenced and modified by each call. For example, in order to hoist a memory reference out of a loop containing a call, RLE needs to know whether the call may change 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.

4.2 Resolving Method Invocations

This section describes techniques for resolving a method invocation site to a monomorphic call which we then replace with a direct call or inline the called procedure. Many techniques for method resolution do not use alias information [Fernandez 1995; Bacon and Sweeney 1996]. Here we describe three straightforward method resolution techniques that do not use pointer analysis—type hierarchy analysis, intraprocedural type propagation, and interprocedural type propagation—and then extend them to use TBAA to analyze pointer dereferences. We use the type hierarchy of Figure 1 as a running example to illustrate the strengths and limitations of the analyses.

4.2.1 *Type Hierarchy Analysis.* Our algorithm for type hierarchy analysis (THA) 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` declared or inherited 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 when `m` is invoked on a variable of type `T`. Since `NULL` is a subtype of all object types 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. If type-hierarchy analysis is used for unsafe languages, such as C++, it may ignore the NULL case.

4.2.2 *Intraprocedural Type Propagation.* Our algorithm for intraprocedural type propagation analysis (TPA) for method resolution is flow-sensitive and uses data-flow analysis to propagate sets of types from *type events* to method invocations

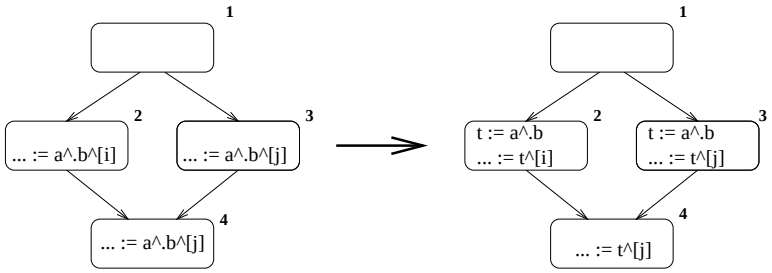

Fig. 7. Eliminating redundant memory loads.

within a procedure. We first present the data-flow equations, and then show an example.

TPA is similar to reaching definitions. In our data-flow lattice, we use a power set of the types; the initial type for a local variable is the empty set. TPA first identifies and propagates sets of possible types for each variable. All nonlocal variables and parameters initially have the maximum set of types consistent with their declaration.[5] In the program, *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. *IsType* is an explicit type discrimination event that checks if v's type is in *Subtypes* (T). *IsType* has two successors, and the appropriate one is picked based on whether or not *IsType* evaluates to *true*. A statement s with a type event generates and kills types as follows:

$$
\begin{aligned}
\textsc{GenType}\ (v \leftarrow \mathit{NEW}(t)) &= \langle v, \{t\}\rangle \\
\textsc{GenType}\ (\mathit{IsType}(v,T)) &= \langle v, \mathit{TypeOf}(v) \cap \mathit{Subtypes}(T)\rangle \text{ for } \mathit{true} \\
&= \langle v, \mathit{TypeOf}(v) - \mathit{Subtypes}(T)\rangle \text{ for } \mathit{false} \\
\textsc{GenType}\ (v \leftarrow u) &= \langle v, \mathit{TypeOf}(u)\rangle \\
\\
\textsc{KillType}(v \leftarrow \mathit{NEW}(t)) &= \langle v, \mathit{TypeOf}(v)\rangle \\
\textsc{KillType}(v \leftarrow u) &= \langle v, \mathit{TypeOf}(v)\rangle
\end{aligned}
$$

T denotes a set of types, t is a single type, and *TypeOf* returns the set of possible types of a variable (for this program point, at this stage of the data-flow analysis). Note that there are two cases for the *IsType* case: one for IsType taking the *true* branch and the other for it taking the *false* branch. The data-flow equations for a statement s are similar to the equations for reaching definitions:

$$
\begin{aligned}
\mathrm{IN}(s) &= \textstyle\bigcup_{p \in \mathrm{PRED}(s)} \mathrm{OUT}(p) \\
\mathrm{OUT}(s) &= \textsc{GenType}(s) \cup (\mathrm{IN}(s) - \textsc{KillType}(s))
\end{aligned}
$$

Our implementation of type propagation propagates types only to scalars; it assumes the conservative worst case (the declared type) for the allocated types of

[5]If a method can be shown to be invoked only via method calls, and not directly as a procedure, then its `self` argument's types can be further restricted to types having this particular method code body as their implementation of a method.

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

Fig. 8. Example to illustrate TPA.

record fields, object fields, array references, and pointer accesses. To demonstrate how TPA works, consider the example in Figure 8.

Statement 2 contains an allocation type event. TPA propagates the type `U` to `o`, and thus determines that the method invocation in Statement 3 calls procedure `mU`. Statement 4 contains an assignment type event, and TPA propagates the type of `p` to `o`, and thus determines that the method invocation in Statement 5 calls procedure `mV`. Finally, TPA merges the types of `o` at the control-flow merge before Statement 6, yielding the type {`U`, `V`} for `o`, and thus cannot resolve `o.m` at Statement 6.

4.2.3 *Interprocedural Type Propagation.* Our algorithm for interprocedural type propagation analysis (ITPA) for method resolution begins by using the results of TPA to build a call graph. The call graph has an edge from a method invocation to each possible target determined by TPA. The algorithm maintains a work list of procedures in depth-first order that need analysis. The work list initially contains all procedures. A procedure needs analysis if new information becomes available about its parameters or about the return value of one of its callees. When ITPA 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. ITPA terminates when the work list is empty.

ITPA 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` has a pointer of type `NULL`, then `error` is invoked instead of this procedure.) ITPA propagates types only to scalars, and it assumes the declared type for all data accessed through pointer traversal. It does not propagate side effects from calls and assigns the declared type for any variable changed by the call. Variables potentially changed by a call include variables declared in outer scopes, globals, parameters passed by reference, and parameter aliases.

ITPA is context-insensitive: rather than analyzing for every combination of call site and callee, ITPA merges 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 (cubic instead of exponential) but at the cost of some accuracy. Consider the following code:

```
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 6.2.2 that, for our benchmark suite, this loss in precision is not significant.

4.2.4 *Using* TBAA *to Resolve Method Invocations.* In this section, we extend TPA and ITPA with TFM-TBAA to obtain TPA-TBAA and ITPA-TBAA, respectively. Whenever TPA-TBAA or ITPA-TBAA encounter a pointer dereference, they invoke TFM-TBAA to get the set of locations referenced by the pointer dereference. TFM-TBAA summarizes this set compactly using type information (e.g., field f of object type O). TPA-TBAA or ITPA-TBAA then propagates the types to or from the set of locations referenced by the pointer dereference. Consider the following code segment:

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

For this example, TPA-TBAA propagates the types of `<rhs>` to the field `f` of all possible objects pointed-to by `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 3.2). Since TFM-TBAA 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 the new type from `<rhs>`. Such updates are called *weak updates* in the pointer analysis literature.

These analyses discover monomorphic uses of general data structures. Consider the linked list package again. When a program links objects of a single type, ITPA-TBAA resolves 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 `T` and the other with type `U`, this analysis does not recognize that each list is homogeneous. It infers the type {`T, U, NULL`} for the elements in both lists. (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 of type `NULL`.)

4.2.5 *Summary and Complexity of Analyses.* Table III 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,

Table III. Summary of Analyses

Analysis	Eliminates NULL	Complexity
THA	No	$O(N_T * \|Methods\|)$
TPA	Yes	$O(\sum_p n_p * v_p)$
ITPA	Yes	$O(N_p \sum_p n_p * v_p)$
TPA-TBAA	Yes	$O(\sum_p (n_p * (v_p + N_T * N_F)))$
ITPA-TBAA	Yes	$O(N_p \sum_p (n_p * (v_p + N_T * N_F)))$

v_p is the number of variables in procedure p, N_T is the number of types in the program, N_F is the maximum number of fields in any type, and N_p is the number of procedures in the program. The complexity for all analyses except for THA is in terms of bit vector steps. The complexity of THA is for one invocation of THA; THA is invoked on demand. These algorithms are simple and therefore fast, as shown in the *Complexity* column.

THA achieves it low time bound because it only examines types and method declarations. TPA achieves its time bound because it is *distributive*, and furthermore *rapid* [Kam and Ullman 1976]. It has the same complexity as reaching definitions for reducible programs; Modula-3 programs are always reducible. TPA stores the possible types of a variable as a set, enabling set union and intersection operations on bit vectors. The length of the bit vectors equals the number of object types in the program, and rarely exceeds 64 in our experience and thus fits entirely inside an integer.

Since ITPA may analyze each procedure multiple times due to recursion and because information flows forward through parameters and backward from return values, it may be substantially slower than TPA. In practice, we have found it to be quadratic in the number of instructions, analyzing each procedure on average 2 to 4 times. Adding TBAA increases the complexity because it propagates types not just to variables but also to aliases which are represented by types and fields in types.

5. METHODOLOGY

In this section, we describe the metrics we used to evaluate TBAA (Section 5.1), our compiler framework (Section 5.2), the benchmark programs we used in the evaluation (Section 5.3), and finally, we discuss how we order the different analyses in the compiler (Section 5.4).

5.1 Metrics

We evaluate TBAA with respect to RLE and method resolution using static and dynamic metrics, and a *limit* analysis. The majority of previous work on alias analysis uses only static properties, such as the size of the *may alias* and *points-to* sets [Banning 1979; Burke et al. 1994; Hind et al. 1999; 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]. A few researchers recently have used dynamic evaluation such as measuring the *execution-time improvement* due

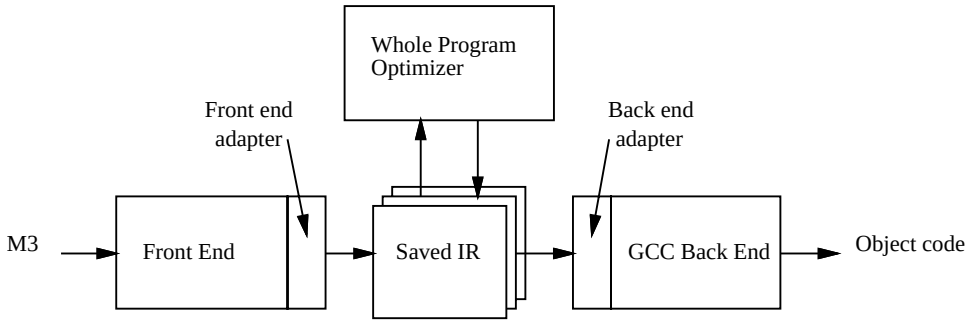

Fig. 9. Compilation framework.

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]. Static, dynamic, and limit evaluation have the following strengths and weaknesses.

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 an alias analysis determines that there are very few aliases, it may not be good enough for an optimization because it fails to disambiguate the key aliases. (2) Static properties do not enable comparisons between the *effectiveness* of two analyses with different strengths and weaknesses. For example, two pointer analyses may report the same number of aliases, but the analyses may disambiguate different pointers and thus enable different optimizations. The main advantage of static evaluation compared to the other metrics discussed below is that it is independent of program runs and inputs.

Dynamic Evaluation. Using dynamic evaluation, such as execution-time improvement, complements static metrics, since execution-time improvements measure the ultimate impact of an analysis (for example, the performance improvement due to pointer analysis and RLE). However, one of their disadvantages is that the results are specific to the given program inputs and to particular uses (such as RLE or method resolution).

Limit Evaluation. Both static and dynamic evaluation have an additional significant shortcoming: these properties do not tell us how much room for improvement there is in the analysis being evaluated except in unusual cases, for example, when an alias analysis disambiguates all memory references. For alias analysis, we would like to know if the aliases really exist at run time, and if any imprecision in the alias analysis causes missed opportunities for optimizations or other clients of the analysis. To detect 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.

Table IV. 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

5.2 Compiler Framework

Figure 9 illustrates our compilation framework which is based on the SRC Modula-3 compiler [Kalsow and Muller 1995]. 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 1989] back end. WPO implements all optimizations and analyses presented in this paper.

5.3 Benchmarks

Table IV describes our benchmarks, and Table V gives the number of noncomment, nonblank lines of code, the number of object types in each benchmark,[6] and the number of method invocations at compile time. For the noninteractive programs, Table V also gives the number of instructions executed, the percent of instructions that are memory loads from the heap, the percent of instructions that are memory loads from the stack and global area (*other*), and the number of method invocations executed at run time. None of these programs were written to be benchmarks, but other researchers have used several of them in their studies [Fernandez 1995; Dean et al. 1996]. Table V contains 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 V do not include instructions or memory references from the standard libraries.

5.4 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 and related transformations, and finally perform RLE. There are interactions between call graph building,

[6]One of the benchmarks, *k-tree*, has object types in generic modules. We only count the number of static object types and not the number of times an object type is instantiated.

Table V. Statistics of Benchmark Programs

Name	Lines	# obj. types	Instructions	% Loads		Method inv.	
				Heap	Other	Static	Dynamic
format	395	10	1,879,195	10	17	37	47,064
dformat	602	12	1,442,541	9	19	95	30,775
write-pickle	654	12	1,614,437	13	16	19	21,251
k-tree	726	3	50,297,517	10	21	13	714,619
slisp	1,645	6	11,462,791	27	9	223	67,253
dom	6,186	70	(interactive)			222	
postcard	8,214	41	(interactive)			293	
m2tom3	10,574	43	50,894,990	8	28	1821	473,559
m3cg	16,475	99	5,636,004	8	21	1808	32,850
trestle	28,977	181	(interactive)			430	

method resolution, and alias analysis, and this process could be iterative or the analyses could be combined. Exploring the interactions between these analyses is beyond the scope of this paper.

6. RESULTS

This section presents the results of evaluating TBAA using the metrics described in Section 5. Since we cannot get reproducible runs for the interactive benchmarks and our dynamic and limit evaluations need multiple runs, we only present results using static metrics for the interactive benchmarks. Section 6.1 presents results evaluating the effectiveness of TBAA for RLE. Section 6.2 presents results evaluating the effectiveness of TBAA for method resolution. Section 6.3 explores the cumulative impact of implementing RLE, method resolution, and inlining. Finally Section 6.4 summarizes our results.

6.1 Evaluation of TBAA Using RLE

Sections 6.1.1, 6.1.2, and 6.1.3 evaluate TBAA with respect to RLE using static, dynamic, and limit evaluations respectively.

6.1.1 *Static Evaluation.* Table VI evaluates the relative importance of the three variations of TBAA: T-TBAA, TF-TBAA, and TFM-TBAA. The table contains the number of static alias pairs determined by each 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 TFM-TBAA is strictly more powerful than TF-TBAA, and TF-TBAA is strictly more powerful than T-TBAA, static metrics are appropriate.

The table shows that TBAA based on field declarations (TF-TBAA) is much more precise than the basic TBAA (T-TBAA), and that selective type merging offers little added precision. Selective type merging reduces the number of intraprocedural

Table VI. Static Alias Pairs as a Percent of All Possible Pairs

	Intraprocedural			Interprocedural		
Program	T-TBAA	TF-TBAA	TFM-TBAA	T-TBAA	TF-TBAA	TFM-TBAA
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

Table VII. Number of Redundant Loads Removed Statically

Program	Loads	T-TBAA (%)	TF-TBAA (%)	TFM-TBAA (%)
format	193	14.0	15.0	15.0
dformat	321	3.1	6.9	6.9
write-pickle	385	11.9	12.2	12.2
k-tree	1018	21.7	22.4	22.4
slisp	1066	3.4	3.5	3.5
dom	3773	8.7	11.2	11.2
postcard	4631	5.6	7.1	7.1
m2tom3	6444	5.7	6.1	6.1
m3cg	6765	7.7	9.1	9.1
trestle	12737	4.1	4.6	4.6

and interprocedural alias pairs for `postcard` and reduces interprocedural aliases for `m3cg`, but 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 intraprocedural aliases are greater than zero), it may be precise enough for some applications.

Table VII evaluates our alias analyses using another static metric: the percent of access paths that RLE removes statically in each of our benchmark programs for each variant of TBAA. The first data column of Table VII (*Loads*) lists the number of static loads in each of the benchmark programs. We only list those loads that are visible to RLE; once a program is compiled to assembly code, it may have more loads than the ones visible to RLE. The next three columns list the number of redundant loads removed by RLE as a percent of total static loads using the three levels of TBAA. Even though RLE does not eliminate redundant loads across procedure boundaries, it does use interprocedural pointer alias information (in the form of mod-ref information); thus, both intraprocedural and interprocedural aliases affect this optimization.

By comparing Table VI and Table VII, we see that the reduction in alias pairs

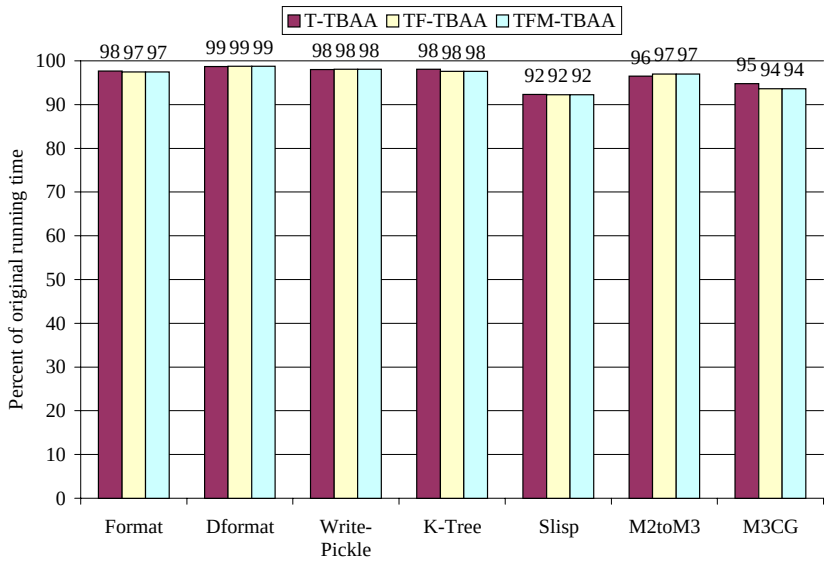

Fig. 10. Impact of RLE.

caused by considering field declarations in TBAA translates into more optimization opportunities: TF-TBAA finds more redundant loads than T-TBAA. The improved precision of selective merges (TFM-TBAA) does not significantly decrease the number of alias pairs, nor increase the number of redundant loads removed.

6.1.2 *Dynamic Evaluation.* This section measures simulated execution-time impact of TBAA on RLE for our noninteractive benchmarks. We measured execution times using a detailed (and validated [Calder et al. 1995]) simulator [Emer et al. 1996] for an Alpha 21064 workstation with one difference: rather than simulating an 8K primary cache we simulated a 32K primary cache to eliminate variations due to conflict misses that we observed in an 8K direct mapped cache. Also, we measured only the execution time spent in user code, since that is the only code that we 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 10 illustrates the simulated execution time impact of TBAA on RLE relative to the original execution time for noninteractive benchmarks. The graph has three bars for each benchmark. Each bar represents the execution time due to RLE and a different alias analysis: T-TBAA (types only), TF-TBAA (types and fields), and TFM-TBAA (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%. One of the benchmarks, *m2tom3*, performs slightly worse with TF-TBAA than with T-TBAA because RLE does not consider register pressure. Note that

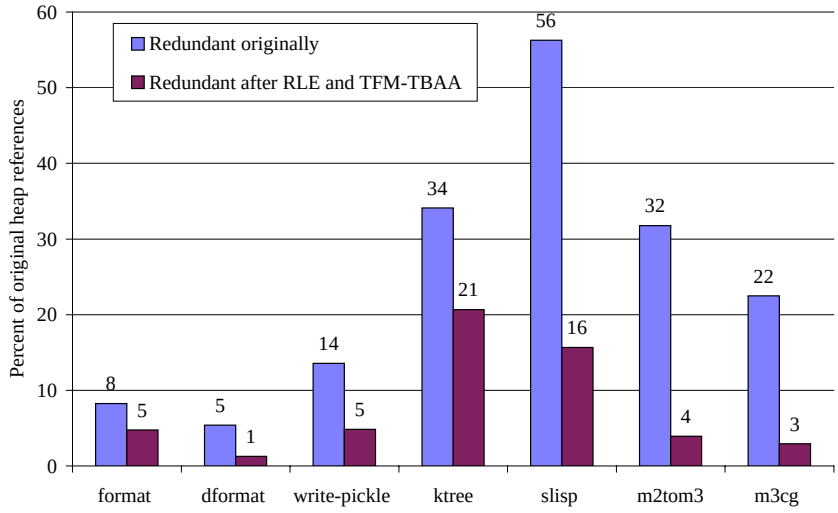

Fig. 11. Comparing TBAA to an upper bound.

the three largest benchmarks benefit the most from RLE. Since RLE is just one of many optimizations that benefits from TBAA, the full impact of TBAA on execution time should be higher. Also, contrary to what the data in Table VI and Table VII 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 analysis 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.

6.1.3 *Limit Evaluation: 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. To discover how effective RLE is, Figure 11 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 11, the bars labeled "Redundant originally" give the fraction of heap references (loads) that are redundant in the original program, and the bars labeled "Redundant after optimizations" give the fraction of heap references that are redundant after TFM-TBAA 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 11 shows that our optimizations eliminate between 35% and 88% of the redundant loads in these programs. Moreover, for 5 of the 7 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. While it is relatively straightforward to expose the hidden loads, it would either lower the level of our intermediate representation or force us to use a multi level intermediate representation.
(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* (PRE) 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**: RLE did not eliminate a redundant load because of an alias that TBAA could not disambiguate.
(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 12 breaks down the *Redundant after Optimizations* bar from Figure 11 into the above five categories. Note that Figure 12 uses a different scale from Figure 11 to make it easier to read. The "alias failure" segment is empty for all the programs and thus not included.

Figure 12 illustrates that *Hidden loads* (dope vector accesses to index open arrays) is the most significant source of the remaining redundant loads. Although, we we did not encounter a single situation when optimization failed because of inadequacies in our alias analysis, there could be some in *Rest*. On average, these loads 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.

6.2 Evaluation of TBAA Using Method Resolution

This section uses static, dynamic, and limit metrics to evaluate the effectiveness of TBAA for method resolution. The bar graphs in this section combine dynamic numbers, represented by the height of the bars, with the corresponding static numbers, written above each bar. Note that we use *site* to refer to *static* measurements, e.g., the number of resolved method invocation sites, and *invocations* to distinguish *dynamic* measurements, e.g., the number of method invocations occurring at resolved

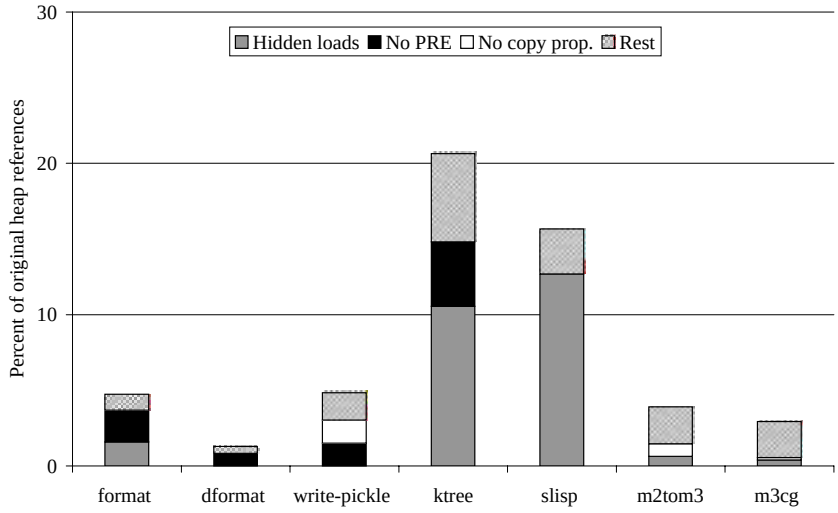

Fig. 12. Source of redundant loads after optimizations.

sites over a run of a program. In these results, we use only the most aggressive version of TBAA: TFM-TBAA. Section 6.3 comments on the results we obtain when we use TF-TBAA.

6.2.1 *Static and Dynamic Evaluation.* Figures 13 through 22 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. The *With NULL* regions in the bars correspond to the percentage of method invocations at run time that analysis resolves to exactly one procedure. The *Ignoring NULL* corresponds to method invocations that analysis resolves to one user procedure or `error`. We obtained these numbers by doing static analyses using each of our method resolution techniques and then scaling the results with the method invocation frequency from a single run of the benchmark; thus, we are also able to provide these numbers for the interactive benchmarks. The pair above the bar is the number of static call sites (With NULL, Ignoring NULL). The *Ignoring NULL* component of the pair includes the *With NULL* component: it is the total number of method resolutions we would resolve if we ignored `NULL`. The pair includes all method invocation sites including ones that may not execute in this execution.

The figures illustrate that type-hierarchy analysis resolves many method invocations for most of the benchmark programs. In addition, the other analyses benefit different benchmarks (though the benefit is not always visible in the dynamic number but rather in the static pairs). TPA resolves very few additional method invocations compared with type hierarchy analysis but removes `NULL` possibilities. Thus, type propagation is useful for languages that have well-defined semantics for

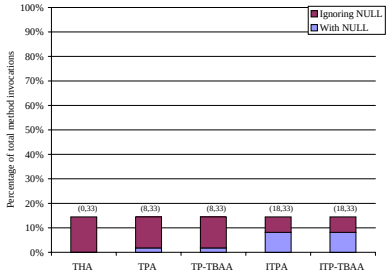

Fig. 13. *format*: Resolved invocations.

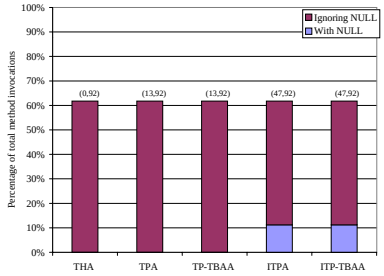

Fig. 14. *dformat*: Resolved invocations.

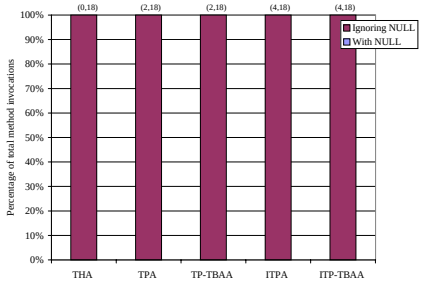

Fig. 15. *write-pickle*: Resolved invocations.

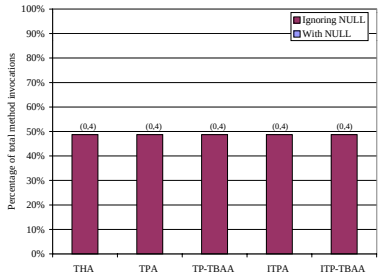

Fig. 16. *k-tree*: Resolved invocations.

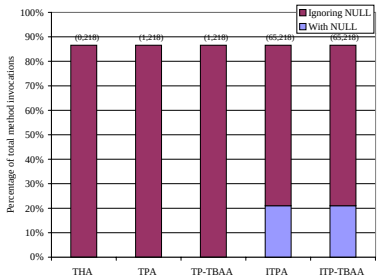

Fig. 17. *slisp*: Resolved invocations.

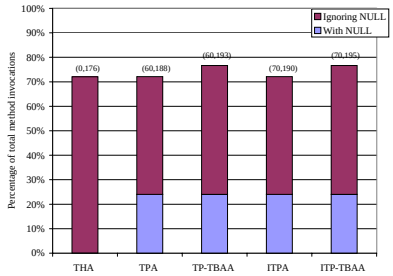

Fig. 18. *dom*: Resolved invocations.

the `NULL` case (such as Modula-3 and Java) but is less useful for other languages (such as C++). TPA-TBAA improves over TPA for two benchmarks, *dom* and *trestle*.

ITPA also eliminates the `NULL` possibility in several of the benchmarks, and resolves additional method invocations (over TPA) in *dom*, *m3cg*, and *trestle*. ITPA-TBAA resolves additional method invocations (over ITPA) 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 benefit from TBAA. The bottom line is that while THA resolves most of the method invocations, other resolution techniques, particularly ones that involve TBAA, are also useful for some benchmark programs, particularly *dom* and *m3cg*.

To judge the execution-time impact of the analyses, we ran our noninteractive

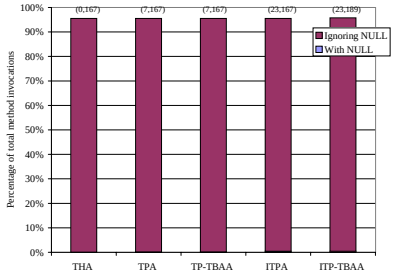

Fig. 19. *postcard*: Resolved invocations.

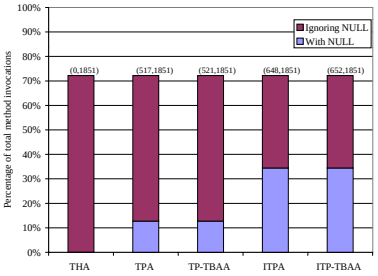

Fig. 20. *m2tom3*: Resolved invocations.

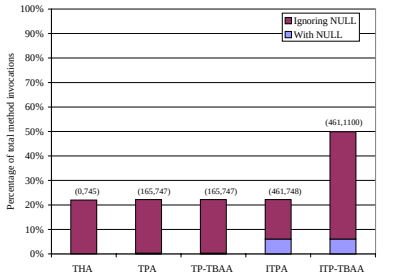

Fig. 21. *m3cg*: Resolved invocations.

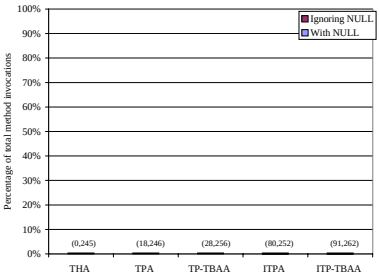

Fig. 22. *trestle*: Resolved invocations.

benchmarks[7] before and after resolution of method invocations on an Alpha 21064 simulator (see Section 6.1.2). 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 NULL* region in Figures 13 through 22. 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 arithmetic mean of 4.6%.

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

6.2.2 *Limit Evaluation.* Programs introduce potential polymorphism by merging control and data as follows:

—Control merges:

[7]Because *trestle*, *postcard*, and *dom* are interactive, we did not include them in this experiment.

Table VIII. Cause of Information Loss

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

—after a conditional statement
—at a call site with multiple targets due to 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 each method invocation that our analyses do not resolve, our cause assignment algorithm finds the *first merge* that results 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.

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 three sources of information loss: data merge, control merge, and code unavailable. *Code unavailable* means that a method could not be resolved due to the unavailability of library code. Table VIII suggests techniques that may prevent the loss of information for each of the three causes of information loss.

Now we address the following questions for the most aggressive version of our method resolution analysis, ITPA-TBAA using TFM-TBAA:

(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 23 answers the first question. Each bar gives the run-time data for one benchmark program. The height of a bar corresponds to the percentage of (dynamic) 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 from sites resolved by analysis, and the "Unresolved" region corresponds to invocations from unresolved monomorphic method sites. The pair above each bar gives the number of static method invocation sites corresponding to the two regions. Note that the numbers above the bar only include those method sites that are executed in our runs. The "Unresolved" region is an upper bound

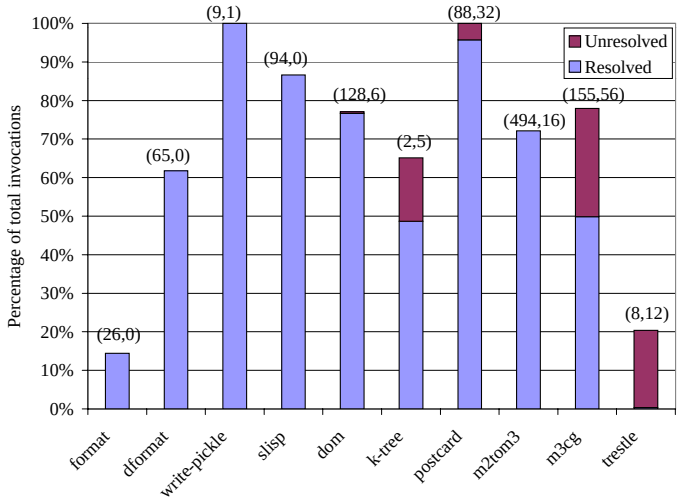

Fig. 23. Monomorphic method invocations.

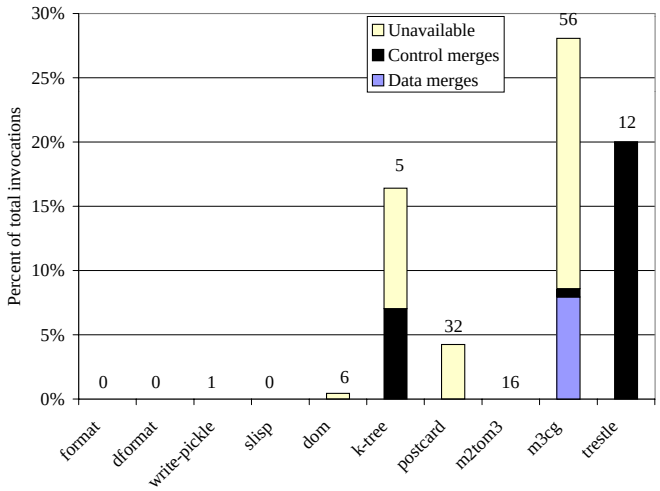

Fig. 24. Monomorphic method invocations that are unresolved.

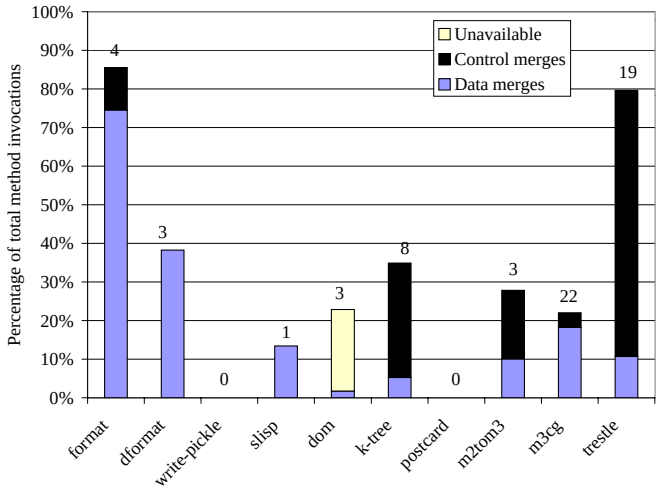

Fig. 25. Polymorphic method invocations.

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 23 shows, that for all benchmarks except *k-tree*, *m3cg* and *trestle*, our analysis resolves the vast majority of monomorphic method invocations; the analyses perform almost as well as the oracle. For the benchmarks where our analyses are less effective, Figure 24 suggests which analyses may be successful in resolving these method invocations.

Each bar in Figure 24 breaks down an *unresolved* region in Figure 23 into three 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. 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. 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.

Figure 25 addresses the second question: what transformations will be effective in converting the polymorphic method invocations to direct calls? Figure 25 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 25 illustrates that most run-time polymorphic method invocations arise because more than one type of object is stored in a heap slot. Two techniques, 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; 1991; Hall 1991].

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 hopefully the code growth induced by transformations such as cloning will be small.

6.3 Cumulative Results

In the previous section we evaluated TBAA with respect to two optimizations: RLE and method resolution. However, these two optimizations are synergistic: method resolution can create new opportunities for RLE, especially if resolved methods are inlined. In this section, we explore this synergy to better understand the full impact of using TBAA for these optimizations.

6.3.1 *Cumulative Execution Time Results.* Figure 26 shows the individual and cumulative impact of method invocation resolution (*Minv*), RLE, and inlining. We present the "base+inlining" column separately so that we can isolate the benefit of inlining resolved method invocations from the benefit of inlining ordinary calls (which does not use any of our analyses). "Minv+RLE+Inlining" should be compared to the "Base+Inlining" bar and not to the original running time. In these experiments, we inlined all direct call sites or resolved method invocation sites that contributed more than 0.8% of the total number of calls in the run. We ran our analyses in the following order: TFM-TBAA, ITPA-TBAA, inlining, and RLE.

This graph shows that our optimizations together have a significant impact on the speed of our benchmark programs. In particular, the *Minv+Rle+inlining* bars show that our two sets of optimizations improve program performance over "Base+Inlining" 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 resolution and RLE is synergistic, i.e., the performance improvement is greater than the sum of the improvements from the three individual optimizations.

In two cases, we observe unexpected slowdown due to the optimizations: *write-pickle* and *slisp*. For *slisp*, method resolution and RLE give significant improvement

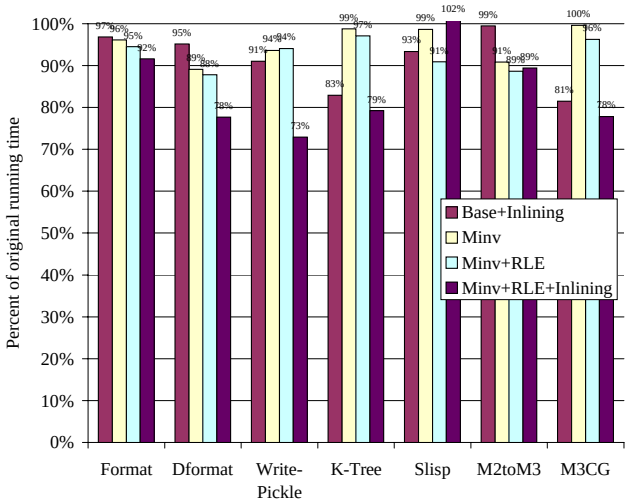

Fig. 26. Cumulative impact of optimizations.

(9%), but doing inlining on top of these optimizations actually slows down the program compared to the "base." When we investigated further we found that the vast majority of the slowdown was due to increased data-cache misses in the inlined version (there was also a small slowdown due to increased instruction-cache misses). We speculate that these misses are caused by inlining a large method, which increased the register pressure and eventually resulted in more data-cache misses. We observed similar behavior with *write-pickle*. Our inliner only considers the frequency of execution when inlining; it should probably consider the size of the procedure as well.

6.3.2 *Cumulative Analysis and Optimization Time.* Table IX gives the analysis time for our most aggressive combination of analyses: ITPA-TBAA using TFM-TBAA. The first data column (*Our optimizations*) column gives the time to perform RLE, method resolution, and TBAA. The second data column (TBAA) gives the approximate time to perform just TBAA. Since part of TFM-TBAA happens on demand when a client requests alias information, we cannot easily separate the TFM-TBAA time from the method resolution time.

In our experiments, we found that TFM-TBAA and TF-TBAA enabled the same optimizations. Because of our implementation of TFM-TBAA and TF-TBAA, we incur much of the overhead of the merging even when we use TF-TBAA. Thus, the analysis times for TF-TBAA are almost identical to TFM-TBAA. The second data column, which gives the total time spent in TBAA, gives a sense for how much maximum improvement we can expect in analysis time if we had implemented TBAA differently so that we did not incur the overhead of merging when we did not need it. The last column (*Time to build*) gives the total amount of time to generate an executable of each program starting from the Modula-3 sources on a 350MHz Alpha 21164 workstation. This time does not include any optimizations or analyses

Table IX. Analysis Time in Seconds for Interprocedural Type Propagation and TFM-TBAA

	Time in seconds		
Program	Our optimizations	TBAA	Build
format	0.2	0.06	17.1
dformat	0.5	0.09	15.4
write-pickle	0.3	0.12	20.0
k-tree	1.1	0.26	23.7
slisp	3.1	0.93	24.0
dom	8.9	1.29	94.4
postcard	10.2	1.80	65.2
m2tom3	32.7	1.44	273.4
m3cg	58.4	6.29	321.9
trestle	43.2	8.10	420.5

described in this paper. TBAA and the optimizations that depend on it increase the total compilation time by only a small percent (up to 15%). However, we should to point out that the Modula-3 compiler is a relatively slow compiler.

6.4 Summary of Results

This section evaluated TBAA using the following different metrics:

(1) Static alias pairs.
(2) Number of opportunities exposed by TBAA for RLE.
(3) Number of method invocations resolved.
(4) Simulated execution-time improvement due to RLE and method resolution.
(5) An upper-bound for TBAA with respect to RLE and method resolution. RLE, method resolution, and inlining.
(6) Analysis time.

Each of these metrics exposes different information about TBAA. The first metric, *static alias pairs*, tells us two things. (1) For our benchmark programs, TFM-TBAA offers little or no improvement in precision over TF-TBAA. (2) TF-TBAA is potentially a much better alias analysis than T-TBAA. Even though TF-TBAA offers little performance improvement over T-TBAA for RLE, it 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 TF-TBAA enables many more opportunities for RLE than T-TBAA. The third metric, *number of method invocations resolved*, reveals that on some programs our techniques resolve the vast majority of method invocations but on several programs (most notably *m3cg* and *trestle*) our analyses fail to resolve the majority of method invocations.

The fourth 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 due to RLE comes from T-TBAA. TF-TBAA improves performance only slightly. The results also illustrate that the execution-time improvement resulting from TBAA and RLE or method res-

olution is relatively small: on average 3.6% improvement for RLE and 4.6% for method resolution.

If we had used only execution-time improvements to evaluate our analysis we might conclude that T-TBAA is the algorithm of choice. However, the *number of opportunities* metric tells us that TF-TBAA is indeed significantly better than T-TBAA. Perhaps with different benchmark inputs TF-TBAA would improve performance significantly more than T-TBAA. If we had used only the *execution-time improvement* metric or the *number of method invocations resolved* metric, 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 method resolution and our benchmark programs.

To summarize, each metric reveals different information about TBAA. For this reason, we feel that static, dynamic, and limit metrics should *all* be used together in a thorough evaluation of an alias analysis, or any compiler analysis for that matter.

7. 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, such as during separate compilation or for Java programs that load classes dynamically. We evaluate the modified analyses by comparing the performance of RLE and method resolution to their performance using the original algorithms.

7.1 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 *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 (though if all code is written in ANSI C++ an alias analysis can make better assumptions about pointers in unavailable code). For type-safe languages such as Modula-3 and Java, the compiler can use type-safety and a type-based alias analysis to make stronger type-safe assumptions about unavailable code. It can assume that unavailable code will not violate the type system of the language. For example, consider the following procedure declaration using the types declared in Figure 2.

```
PROCEDURE f (p:  REF S1; q:  REF 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` may point to the same object. For a type-safe language, a type-based analysis can safely assume that `p` and `q` cannot point to the same object 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

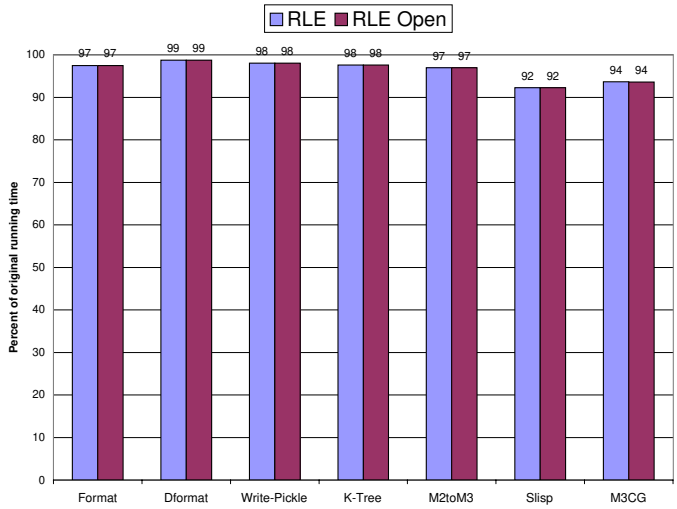

Fig. 27. Simulated execution time using open- and closed-world assumptions

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 TFM-TBAA more conservative at merges. We merge any two types (related by the subtype relation) to which the program 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 27 compares the simulated running time improvement resulting from RLE when assuming that the entire program is available (closed world) and assuming it is not available (open world). 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 VII, since TFM-TBAA, which is most affected by the open-world assumption, does not enable any additional opportunities for RLE over TF-TBAA. With respect to the static metrics, we found that they were the same

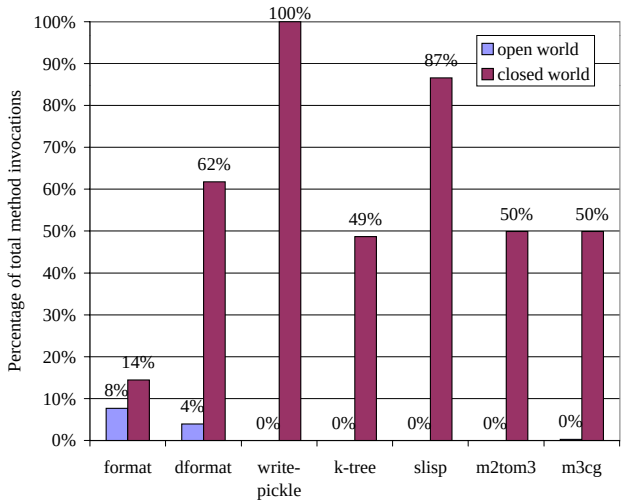

Fig. 28. Percent of resolved method invocations with open- and closed-world assumptions.

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. These additional alias pairs did not reduce the effectiveness of RLE.

We also need to modify method resolution analyses if the entire program is not available for analysis. If some of the assignments and type hierarchy are unavailable for analysis, 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 nonlocal 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 28 compares the percent of dynamic method invocations out of all method invocations that our analysis can resolve assuming the entire program is available (closed world) and assuming some portion is unavailable (open world). The open-world assumption dramatically limits the number of method calls that our analysis resolves.

8. APPLICABILITY TO OTHER OPTIMIZATIONS AND LANGUAGES

This paper has demonstrated that TBAA works for two specific optimizations that both can benefit from locally precise information. We believe this property will make it effective for other scalar optimizations such as dead-code elimination, constant propagation, scheduling, and register allocation. This speculation needs further testing of course.

The analyses described here are language independent, but their usefulness depends on both language and 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: Reinig [1998] in their DEC C++ compiler and Nystrom et al. [1999] in their Java optimizer. We discuss these further in related work.

While Java programs are type safe, they introduce a different set of challenges for TBAA and associated optimizations. In particular, the exception model, memory model, and threads severely limit the extent to which an optimization can reorganize code [Nystrom et al. 1999].

The effectiveness of our method resolution analyses depends on programming style and type-safety as well. For example, some C++ programming styles discourage the use of virtual functions unless necessary;[8] in essence, this 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 if these programs use multiple inheritance. Since there are no static types in dynamically-typed languages, our results will not directly apply to them.

9. RELATED WORK

In this section, we distinguish our work from others that address alias analysis, method resolution, and compiler optimization evaluation. For alias analyses, we focus on those papers that present algorithms similar to ours or evaluate alias analyses using more than static metrics.

9.1 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; Hind et al. 1999; 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; Hummel et al. 1994; Cooper and Lu 1997; Larus and Hilfinger 1988; Wilson and Lam 1995]. The key differences between the algorithms stem from 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 three ways: (1) It is type-based instead of instruction-based. (2) We evaluate our alias analyses with respect to two optimizations, RLE and method resolution, rather than using static measurements as used by most work on alias analysis [Banning 1979; Burke et al. 1994; Hind et al. 1999;

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

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]. (3) For both our optimizations that benefit from alias analysis, we use a limit study to demonstrate that TBAA is close to perfect for our benchmarks and optimizations. Our limit studies are similar to those of Wall [1991], which assumes a "perfect alias analysis" to find an upper bound on instruction-level parallelism. 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. [1986] and Chase et al. [1990] were among the first to write that using programming language types could improve alias analysis, but did not present algorithms that did so and did not evaluate it. Our alias analysis is most similar to those of Rinard and Diniz [1996], Steensgaard [1996], and Ruf [1995; 1997].

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 [1996] presents an instruction-based alias algorithm that uses nonstandard types, not programming language types, to obtain a nearly linear-time 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. In terms of precision, Steensgaard's algorithm is not directly comparable to TBAA, and there are many examples where Steensgaard does better or worse than TBAA.

Ruf [1995] compares a context-sensitive alias analysis to a context-insensitive one and finds, for his benchmarks, that they are comparable in precision. Both algorithms are flow-sensitive and are fairly simple versions of context-insensitive and -sensitive algorithms in that they do not consider any shape information (such as Chase et al. [1990]). Both algorithms considered by Ruf are more precise than TBAA, since they are flow-sensitive and also support strong updates. Ruf finds that there is little difference for his benchmarks between context-sensitive and context-insensitive versions of his analyses. Our work suggests that the point of diminishing return for pointer analyses may come even earlier for many applications than Ruf's context-insensitive analysis.

Ruf [1997] shows how to use programming language types and nonstandard types (such as those of Steensgaard [1996]) 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. In Ruf's first algorithm, he uses only dependences between programming language types; thus the kind of type information he uses is similar to T-TBAA. Ruf uses his scheme to partition programs for alias analyses, but does not use programming language types in the analysis. Ruf's second algorithm does not use programming language types; instead it uses nonstandard types (e.g., those of Steensgaard [1996]).

Wilson and Lam [1995] present a context- and flow-sensitive pointer analysis for C programs. This analysis handles the entire C language and is thus quite complex. Wilson and Lam introduce the use of partial-transfer functions for pointer analysis, which allow even their context-sensitive analysis to reuse prior analyses of procedures. Wilson and Lam evaluate their algorithm using static metrics and one

dynamic metric: the speedup due to automatic parallelization of two C programs which could previously not be fully parallelized because of how they used pointers. This analysis is much more powerful than any TBAA but is not always practical even for modestly sized programs [Wilson 1997].

Cooper and Lu [1997] describe and evaluate register promotion, an optimization that moves memory references out of loops and into registers. Register promotion, when it includes the extension for pointer-based loads, is similar to the loop-invariant code motion part of RLE except that promotion also hoists stores out of loops and not just loads. They evaluate register promotion with two alias analyses: a trivial analysis and a flow-sensitive alias analysis. Their flow-sensitive analysis is similar to the context-insensitive analysis of Ruf [1995]. They used the number of instructions executed as their performance metric and found that the more powerful alias analysis did not significantly improve performance. We observe more performance improvement due to RLE, which may be because we measure object-oriented programs as opposed to the C programs used by Cooper and Lu. Calder et al. [1994] show that C programs typically execute a smaller percentage of loads and stores than C++ programs.

Debray et al. [1998] describe an alias analysis for executable code. They evaluate their algorithm by measuring the percentage of loads eliminated using loop-invariant code motion and PRE of loads. They do not present execution time improvements or a limit study for their alias analysis.

Shapiro and Horwitz [1997a] evaluate the impact of four flow-insensitive alias analyses on a range of applications. The four alias analyses are naive, Steensgaard [1996], Anderson [1994], and their own alias analysis [Shapiro and Horwitz 1997b], whose precision is approximately between Steensgaard's and Anderson's. With the exception of "naive," which is weaker than TBAA, the other analyses are incomparable with TBAA. It is easy to contrive examples that show the superiority of one over the other. For instance, unlike TF-TBAA or TFM-TBAA, Shapiro and Horwitz's algorithms do not separate fields in their analyses. Shapiro and Horwitz compare the pointer analyses by counting optimization opportunities rather than the performance impact of the optimizations.

Ghiya and Hendren [1998] use their pointer analysis, called *connection analysis*, to improve scalar optimizations, particularly loop-invariant removal, location-invariant removal, and common-subexpression elimination, and present running time improvements. The combination of loop-invariant removal and common-subexpression elimination is similar to RLE. Connection analysis is a very weak pointer analysis, but since it is flow-sensitive, in some cases it may be more powerful than TBAA. Their paper evaluates connection analysis by measuring the number of opportunities for their optimizations and by measuring the running time performance improvement that results. They do not present a limit study.

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.

Since the first publication of some of our algorithms [Diwan et al. 1998], two groups have applied TBAA to other languages. 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, yields 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 type-safe C programs.

Nystrom et al. [1999] apply the "incomplete program" version of TBAA to Java programs in a bytecode-to-bytecode optimizer and use it for intraprocedural PRE of memory references. PRE of memory references is more powerful than RLE in that it can eliminate not just fully redundant memory references but also partially redundant ones. They apply their optimization only intraprocedurally (and using only intraprocedural information) since any call can potentially result in a thread switch. They get execution time improvements of up to 9% (but usually much less—average 1%) for their programs. They find that Java's exception model significantly hinders their ability to optimize Java programs.

9.2 Other Related Work on Method Invocation Resolution

Fernandez [1995] and 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. [1996] describe and evaluate a range of transformations and analyses for resolving method invocations in object-oriented languages. Their paper combines many of the ideas in other papers discussed in this Related Works section; it also serves as an excellent overview of the area. Specifically, Chambers et al. describe class hierarchy analysis and an analysis similar to TPA (called intraprocedural class analysis). They do not evaluate these algorithms using a limit study and do not study the impact of pointer analyses on method resolution.

Palsberg and Schwartzbach [1991], Agesen and Hölzle [1995], and Plevyak and Chien [1994] describe *type inference*[9] 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 [1992], Calder and Grunwald [1994], Hölzle and Ungar [1994], Dean et al. [1994], and Grove et al. [1995] describe transformations for converting polymorphic method invocations to direct calls, which we did not perform. 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 [1996] and Aigner and Hölzle [1996] evaluate techniques for resolving method invocations in C++ programs. Bacon and Sweeney evaluate three fast analyses, including type hierarchy analysis and *rapid type analysis* (RTA), for resolving method invocations in C++ programs. Bacon and Sweeney also use a limit study to evaluate their analyses.

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

Bacon and Sweeney evaluate flow-insensitive analyses. Aigner and Hölzle evaluate type feedback and type hierarchy analysis and find that they are both effective at resolving method invocations. Our analysis is flow-sensitive and uses alias analysis, and is thus more precise.

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.

Shivers [1991] describes and classifies a range of analyses to discover control flow in Scheme programs. Our interprocedural type propagation and 0CFA are both context-insensitive. However, Shivers's analysis is optimistic with respect to the call graph while ours is pessimistic. 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 [1995] algorithm performs pointer analysis at the same time as method invocation analysis. Plevyak and Chien's [1994] type inference algorithm also does some pointer analysis. Both algorithms are flow-sensitive and at least somewhat context-sensitive and are thus more powerful than TBAA but much 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 (median 6 seconds, with a number of larger benchmarks than theirs). Subsequent work [Chatterjee and Ryder 1997a; 1997b; Chatterjee et al. 1999] improves the scalability of their analyses. We show, that for our benchmarks and optimizations, our simple analyses are effective, and that there is little to be gained by more powerful analyses. This result originates in part from Modula-3's language semantics, which restricts aliasing; a more powerful alias analysis may be more useful for C++ than for Modula-3, but to our knowledge this need has not yet been demonstrated for significant applications.

DeFouw et al. [1998] describe a parameterized framework that integrates a range of analyses for method resolution. This framework can encompass fast and simple analyses such as RTA [Bacon and Sweeney 1996], Steensgaard-like analyses [Steensgaard 1996], and 0-CFA [Shivers 1991] (which, as discussed above, is a more precise version of ITPA). DeFouw et al. use this framework to evaluate a range of analyses, including those just mentioned and some new analysis opportunities that their framework exposes. For their evaluation they use several static and dynamic metrics, including number of method invocations resolved and execution speedup. They do not use any limit study for their evaluation. They find that for Java programs there is little or no difference between the different analyses. However, for Cecil programs there is a significant difference between the analyses for the small programs and modest difference between the larger programs. Even on the larger programs, they get most of their benefit from the simpler analysis. Our results support theirs: for many applications, a fast and simple alias analysis may be sufficient.

A key difference between our work and that of all 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.

9.3 Evaluating Optimizations

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.

10. CONCLUSIONS

We described and evaluated three algorithms that use programming language types to disambiguate memory references. The first analysis, T-TBAA, uses type compatibility to determine aliases. The second, TF-TBAA, extends the first by using additional high-level information such as field names and types. The third, TFM-TBAA, extends the second with a flow-insensitive analysis. We show that the algorithm that uses only type compatibility gives the vast majority of performance improvement though the other two analyses improve on it with respect to the static metrics (and thus may yield greater performance improvements for other programs or runs). We evaluated these pointer analyses with respect to two clients of pointer analysis: redundant load elimination (RLE) and method resolution.

TBAA with 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.

TBAA with method resolution is quite effective. On average, our analyses resolve more than 92% of the method invocation sites that are amenable to analysis. Applying method resolution and inlining improves the running time of the benchmark programs by up to 11%. Combined with RLE, the improvements are even higher. For method invocations that are unresolved by our analyses, we determine the reason for analysis failure. We find that for the most part TBAA is precise for method resolution, but for some programs, a more precise alias analysis may be justified. Finally, TBAA with method resolution performs much worse with the open-world assumption than with the closed-world assumption.

In summary, we have shown that simple, fast type-based analyses are an effective tool for optimizing object-oriented programs, and for a selection of optimizations, they are close to perfect.

ACKNOWLEDGMENTS

We would like to thank Ole Agesen and Darko Stefanović for comments on drafts of this paper. We would like to thank the anonymous referees for their detailed comments.

REFERENCES

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, 142–166.

Anderson, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU.

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, 324–341.

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. 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.

Calder, B., Grunwald, D., and Zorn, B. 1994. Quantifying behavioral differences between C and C++ programs. Tech. Rep. CU-CS-698-94, University of Colorado, Boulder, CO. Jan.

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. 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. 1991. Making pure object oriented languages practical. In *Proceedings of the ACM SIGPLAN '91 Conference on Object-Oriented Programming Systems, Languages, and Applications*. 1–15.

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, 133–146.

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, 308–319.

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 Twenty Fifth 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.

Diwan, A., McKinley, K. S., and Moss, J. E. B. 1998. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*. Montreal, 106–117.

Diwan, A., Moss, E., and McKinley, K. S. 1996. Simple and effective analysis of statically typed object-oriented programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Object-Oriented Programming Systems, Languages, and Applications*. San Jose, CA, 292–305.

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, 306–323.

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.

Emer, J., Webb, D., and McCallig, M. 1996. Zippy simulator for alpha workstations. Software.

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 Twenty Fifth Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. 121–133.

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.

Hind, M., Burke, M., Carini, P., and Choi, J.-D. 1999. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems 21,* 4 (July), 848–894.

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.

Kalsow, B. and Muller, E. 1995. *SRC Modula-3 Version 3.5*. Systems Research Center, Digital Equipment Corporation, Palo Alto, CA.

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.

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)*.

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, 54–67.

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 *Conference Record of the Twenty Fourth Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. Paris, France, 15–26.

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 Twenty Fourth Annual ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*. Paris, France, 1–14.

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. 1989. *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 Twenty Third 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.

Tarjan, R. E. 1975. On the efficiency of a good but not linear set union algorithm. *Journal of the ACM 22,* 2, 215–225.

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. 1997. Efficient context-sensitive pointer analysis for C programs. Ph.D. thesis, Stanford University, Palo Alto, CA.

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.

Received August 1999; revised June 2000; accepted January 2001