Efficient Predicate Dispatching

Craig Chambers and Weimin Chen

Department of Computer Science and Engineering
University of Washington

Technical Report UW-CSE-98-12-01

Abstract

The speed of method dispatching is an important issue in the overall performance of object-oriented programs. We have developed an algorithm for constructing efficient dispatch functions for the general *predicate dispatching* model, which generalizes single dispatching, multiple dispatching, predicate classes and classifiers, and pattern-matching. Our algorithm generates a *lookup DAG* each of whose nodes represents an *N*-way test of the class or value of a formal or other expression. Our algorithm implements each of these *N*-way tests with a *binary decision tree* blending class identity tests, class range tests, and table lookups. Our algorithm exploits any available static information (from type declarations or class analysis) to prune unreachable paths from the lookup DAG, and uses any available dynamic profile information to minimize the expected time to traverse the binary decision trees. We measure the effectiveness of our dispatching algorithms on a collection of large Cecil and Java programs, compiled by the Vortex optimizing compiler, showing improvements of up to 40% over already heavily optimized baseline versions.

1 Introduction

The speed of method dispatching is an important issue in the overall performance of object-oriented programs. Previous work on techniques for efficient dispatching has typically focused on singly dispatched languages; a handful of techniques have been proposed for multiply dispatched languages, and almost none for languages with other kinds of dispatching semantics such as predicate classes [Chambers 93b] or classifiers [Hamer et al. 90, Mugridge et al. 91]. Additionally, previous work has focused on applying particular techniques universally to all dispatches in a program.

Recently, a more general model of dispatching, *predicate dispatching*, was introduced [Ernst et al. 98]. Predicate dispatching associates each method with a *dispatching predicate*, a boolean formula over subclass and value tests of the method's formals and expressions derived from the formals. A method is applicable to a given message if its dispatching predicate evaluates to true after binding the message's actuals to the method's formals, and one method overrides another if the first's predicate logically implies the other's. Static typechecking can guarantee that a program can have no message-not-understood or message-ambiguous errors at run-time. Predicate dispatching includes most previous dispatching mechanisms as special cases, including single dispatching, symmetric multiple dispatching, predicate classes, classifiers, modes, and pattern-matching as found in most functional programming languages, shedding light on the relationships between these mechanisms. Additionally, new kinds of dispatching idioms and new combinations of existing idioms are possible with predicate dispatching. However, because of its greater expressive power, it was not clear how to implement predicate dispatching efficiently.

In this paper we present an algorithm for constructing efficient dispatch functions for the predicate dispatching model. The generality of the predicate dispatching model does not reduce the effectiveness of our algorithm on the special cases, however; for multiple dispatching and predicate classes, our dispatch functions are more efficient than previously described dispatch techniques. Our algorithm consists of two main components, one computing the high-level structure of the dispatch function and the other computing the low-level implementation of the individual

single-dispatches chosen by the high-level component. The high-level algorithm takes the set of methods in a particular generic function (collection of dynamically overloaded methods with the same name, number of arguments, and static argument types) and any available static information about the possible classes of arguments and generates a *lookup DAG*, a rooted directed acyclic graph. Each interior node in the lookup DAG represents a test of the class or value of a formal or derived expression, with the node's *N* outgoing edges representing the different possible outcomes of the test. Each leaf node represents the target method (or system-generated error method) to invoke for the various combinations of test outcomes that lead to that leaf. Each interior node acts like a single dispatch, and overall dispatching is made up of combinations of these individual single dispatches. Compared to previous algorithms for multimethod dispatching, our algorithm handles the more general predicate dispatching model, can select an appropriate order in which to test the classes of dispatched formals (not just left-to-right), can skip tests of formals along those paths where their dynamic class does not affect the outcome of dispatching (not just testing all formals along all paths), and can exploit available static information to skip tests whose outcomes are statically determined.

Each interior node in the lookup DAG conceptually represents an *N*-way branch on a single value or class. This single dispatch can be implemented in many different ways; different techniques may be best for different single dispatches, depending on the number of target nodes, the number, spread, and relative likelihood of possible values of the expression being tested, the execution frequency of the test itself, and the relative importance of dispatcher space versus dispatch speed. In fact, some combination of techniques may be best for a given single dispatch. Our algorithm implements each of these *N*-way tests with a binary decision tree blending class identity tests, class identifier inequality tests, and one-dimensional array lookups. The appropriate blend of these three kinds of tests is guided by dynamic profile information or static estimates of the relative likelihood of the possible classes or values of the expression being tested.

We have implemented several variations on our algorithm in the context of the Vortex optimizing compiler [Dean et al. 96, Chambers et al. 96], and we applied our algorithm to generate dispatch functions at link-time, one dispatcher per generic function. For a set of large benchmark programs written in Cecil [Chambers 92, Chambers 93a] (a purely object-oriented language including multimethods and predicate classes, a large subset of the predicate dispatching model) and Java [Gosling et al. 96] (a hybrid language with only single dispatching), we measured the speed of execution and the space consumed by dispatch functions, comparing different variations of our algorithm and the previous dispatch implementation based on dynamically generated call-site-specific polymorphic inline caches [Hölzle et al. 91]. We observed speed-ups in Cecil programs of up to 30%, with greater speed-ups for larger programs, and speed-ups for Java programs of up to 40%. In all our experiments, we applied Vortex's full range of optimizations, so that the benefits we observe for our dispatch algorithm are in addition to the benefits obtained by optimizing dispatches through other optimizations.

In the next section, we describe the predicate dispatching model. In section 3 we describe our algorithm for determining the high-level organization of the lookup DAG, and compare our algorithm to related work at this level. In section 4 we describe our algorithm for determining the low-level implementation of each node of the lookup DAG, and compare our algorithm to related work at this level. Section 5 presents our experimental assessment of the time and space costs of our algorithm. We conclude with a discussion of contributions and future work in section 6.

2 Model of Dispatching

In the predicate dispatching model, a *generic function* includes a collection of dynamically overloaded methods. For simplicity, we assume all the methods use the same names for their formals, given as part of the generic function

declaration. Each method is defined by a predicate expression giving the method's applicability and specificity and a method body specifying the code to run if the method is invoked. The following grammar defines the structure of a generic function, as viewed by the compiler after collecting together all the methods in that generic function in the program and renaming the formal parameters of all methods in the generic function to be the same:

```
GF     ::= gf Name(Name1,...,Namek) Method1 ... Methodn
Method ::= when Pred { Body }
Pred   ::= Expr@Class          test whether Expr is an instance of Class or a subclass
         | test Expr           test whether Expr (a boolean-valued expression) is true
         | Name := Expr        bind Name to Expr, for use in later conjuncts and the method body
         | not Pred            the negation of Pred
         | Pred1 and Pred2     the conjunction of Pred1 and Pred2
         | Pred1 or Pred2      the disjunction of Pred1 and Pred2
         | true                the always-true default predicate
Expr   ::= host language expression; assumed to have no externally visible side-effects
Class  ::= host language class name
Name   ::= host language identifier
```

When a generic function is applied to a tuple of argument objects, the predicate of each method in the generic function is evaluated, in an environment binding the generic function's formals to the argument objects, to determine if that method applies to the arguments. The different kinds of predicates are evaluated as follows:

- `Expr@Class` is true iff evaluation of `Expr` in the current environment yields an object that is an instance of `Class` or some subclass of `Class`.
- **`test`** `Expr` is true iff evaluation of `Expr` in the current environment yields the `true` object.
- `Name` **`:=`** `Expr` is always true, but updates the current environment to bind `Name` to the result of evaluating `Expr`.
- **`not`** `Pred` is true iff `Pred` is not. Any updates to the environment by `Pred` are ignored.
- $\texttt{Pred}_1$ **`and`** $\texttt{Pred}_2$ is true iff $\texttt{Pred}_1$ is true (yielding an enhanced environment if $\texttt{Pred}_1$ contains any name bindings) and $\texttt{Pred}_2$ is true in the enhanced environment. $\texttt{Pred}_2$ is only evaluated if $\texttt{Pred}_1$ is true, allowing expressions in $\texttt{Pred}_2$ to be defined only for cases where $\texttt{Pred}_1$ succeeds. For example, **`test`** `x != 0` **`and test`** `y/x > 5` exploits the required evaluation order, as does `list@Cons` **`and`** `list.head@Int`.
- $\texttt{Pred}_1$ **`or`** $\texttt{Pred}_2$ is true iff $\texttt{Pred}_1$ is true or $\texttt{Pred}_2$ is true. There are no constraints on the evaluation order of $\texttt{Pred}_1$ and $\texttt{Pred}_2$, and any updates to the environment by $\texttt{Pred}_1$ or $\texttt{Pred}_2$ are invisible outside of $\texttt{Pred}_1$ or $\texttt{Pred}_2$, respectively.
- **true** is always true.

After collecting the set of applicable methods, the unique most-specific method is identified and then invoked. One method m_1 is deemed at least as specific as another method m_2, written $m_1 \leq_{\texttt{Method}} m_2$, exactly when m_1's predicate implies m_2's. The different cases in which one predicate implies another, determined statically based on the structure of the two predicates, are as follows:

- $\texttt{Expr}_1\texttt{@Class}_1$ implies $\texttt{Expr}_2\texttt{@Class}_2$ iff $\texttt{Expr}_1$ is the same as $\texttt{Expr}_2$ and $\texttt{Class}_1$ is equal to or a subclass of $\texttt{Class}_2$. Two expressions are the same if their abstract syntax trees are isomorphic; this conservative definition of equivalence retains decidability of predicate dispatching.
- **`test`** $\texttt{Expr}_1$ implies **`test`** $\texttt{Expr}_2$ iff $\texttt{Expr}_1$ is the same as $\texttt{Expr}_2$.
- Since it is always true, any predicate implies `Name := Expr`.
- **`not`** $\texttt{Pred}_1$ implies **`not`** $\texttt{Pred}_2$ iff $\texttt{Pred}_2$ implies $\texttt{Pred}_1$.
- $\texttt{Pred}_1$ **`and`** $\texttt{Pred}_2$ implies both $\texttt{Pred}_1$ and $\texttt{Pred}_2$.
- A predicate `Pred` implies both `Pred` **`or`** $\texttt{Pred}_2$ and $\texttt{Pred}_1$ **`or`** `Pred`.

- Any predicate implies **true**.

(To be complete up to equivalence of expressions, predicates should be converted into disjunctive normal form before applying the rules for `not`, `and`, and `or`.) It is a message-not-understood error if no applicable methods are found, and it is a message-ambiguous error if no applicable method is the unique most-specific one. A sound and complete static typechecking algorithm exists for determining whether invocation of any given generic function can ever lead to a message-not-understood or message-ambiguous error. Formal rules for the static and dynamic semantics of the predicate dispatching model are given in the earlier paper introducing predicate dispatching [Ernst et al. 98].

The predicate dispatching model includes most existing dispatch mechanisms as restricted cases, and consequently our algorithm for constructing efficient dispatchers applies to all these other models. Every object-oriented language includes the notion of generic function, either explicitly (as in Common Lisp [Bobrow et al. 88, Steele Jr. 90] and Dylan [Shalit 96]) or implicitly (as in Cecil, Smalltalk [Goldberg & Robson 83], C++ [Stroustrup 91], Java, Eiffel [Meyer 92], and Modula-3 [Nelson 91]). In Cecil and Smalltalk, a generic function corresponds to all the methods having the same message name and number of arguments. In C++, Java, Eiffel, and Modula-3, a generic function roughly corresponds to all the methods having the same name, number of arguments, and static argument types; more precisely, a generic function is created whenever a class contains a method declaration that doesn't override any inherited method declaration, and all methods that override the introducing method declaration are included in that method's generic function. Each dynamically dispatched call site applies a particular generic function to its arguments, selecting the single most-specific method in the generic function to invoke.

Most previous dispatching models correspond to special idiomatic uses of predicates:

- With single dispatching, a method `m` in a class $\texttt{C}_\texttt{m}$ would be modeled with a predicate of the form $\texttt{self@C}_\texttt{m}$, where `self` is the name of the generic function's first formal.
- With multiple dispatching, a multimethod `m` with `k` arguments that specializes its `i`'th formal $\texttt{formal}_\texttt{i}$ to the class $\texttt{C}_\texttt{mi}$ would be modeled with a predicate of the form $\texttt{formal}_1\texttt{@Class}_{\texttt{m1}}$ **and** ... **and** $\texttt{formal}_\texttt{k}\texttt{@Class}_{\texttt{mk}}$. Omitting a $\texttt{formal}_\texttt{i}\texttt{@Class}_{\texttt{mi}}$ conjunct leaves that argument position unspecialized for that method; different methods can specialize on different argument positions.[1]
- With predicate subclasses [Chambers 93b], a predicate subclass `PredClass` can be declared that is in force whenever instances of its superclass `Class` satisfy some additional boolean predicate `Test` (evaluated in an environment where the identifier `Class` is bound to the instance being tested). A method `m` whose formal $\texttt{formal}_\texttt{i}$ specializes on the predicate class `PredClass` can be modeled by replacing $\texttt{formal}_\texttt{i}\texttt{@PredClass}$ with $\texttt{formal}_\texttt{i}\texttt{@Class}$ **and test** `Test'`, where `Test'` is the same as `Test` except that free references in `Test` to `Class` are changed to $\texttt{formal}_\texttt{i}$. Classifiers [Hamer et al. 90, Mugridge et al. 91] and modes [Taivalsaari 93] can be modeled with similar techniques [Ernst et al. 98].
- With pattern-matching, as found in languages like ML [Milner et al. 97] and Haskell [Hudak et al. 92], a function case can be defined for certain arguments based on their value or datatype structure, possibly examining the value or structure of subcomponents. Conjunctions can be used to test the properties of multiple arguments, as with multimethods. Tests on the arbitrarily nested subcomponent *`componentReferencePath`* of an argument $\texttt{formal}_\texttt{i}$ can be modeled by using $\texttt{formal}_\texttt{i}\texttt{.}$*`componentReferencePath`* as the expression being tested. Tests for an expression `Expr` having a particular value `Value` can be modeled either by adopting a prototype-based language model (in which case "values" and "classes" are tested uniformly using `Expr@Value` clauses),

[1] This model for multimethods applies to languages like Cecil and Dylan that treat all argument positions uniformly, since implication between conjuncts is symmetric. Languages like Common Lisp that prioritize earlier arguments over later arguments are not supported directly by the predicate dispatching model.

or by using the conjunction `Expr@Class`$_{\texttt{Value}}$ **`and test`**`(Expr = Value)`, where `Class`$_{\texttt{Value}}$ is the class of `Value`. `Name := Expr` bindings can be used to give names to subcomponents, for use in the body of the method. (Compared to pattern-matching like that in ML and Haskell, predicate dispatching confers the additional benefits of inheritance of cases from superclasses to subclasses, the ability to add new cases to handle new subclasses without modifying existing cases or declarations, automatic ordering of cases based on specificity of patterns as opposed to textual order of cases, and the ability to reference a bound variable in later patterns (non-linear patterns)).

- Boolean guards on patterns, as found in Haskell, correspond to additional **`test`** predicates, potentially over multiple formals and local bindings.

In addition to modeling many previous dispatching mechanisms, new kinds of dispatching can be specified under predicate dispatching, including general disjunctions and negations of tests as well as combinations of primitives not previously supported, such as subclass testing combined with testing of arbitrary subcomponents.

3 Multiple Dispatch Implementation

Given a generic function, our algorithm first determines the high-level strategy for selecting the right method to invoke, encoding its decisions in a *lookup DAG*. Any available information about the possible classes of the arguments to the generic function, such as determined by static type or class declarations [Johnson 86] or static class analysis [Johnson et al. 88, Chambers & Ungar 90, Plevyak & Chien 94, Fernandez 95, Agesen 95, Dean et al. 95, Grove et al. 97, DeFouw et al. 98], is exploited to produce a faster, smaller lookup DAG.

The next subsection defines the lookup DAG structure. Subsections 3.2 through 3.6 present the steps in our algorithm to construct the lookup DAG. Subsection 3.7 compares our algorithm with other algorithms for multimethod dispatching. Throughout this section we use the running example shown in Figures 1 and 2.

3.1 The Lookup DAG

A lookup DAG $G=(N, E\subseteq N\times N, n_0\in N)$ is a rooted, directed acyclic graph, representing a decision "tree" but with identical subtrees shared to save space. Each interior node $n\in N$ in the lookup DAG has a set of outgoing edges $n.edges=\{(n,n')\in E\}$ and is labeled (*n.expr*) with an expression `Expr`$_{\texttt{i}}$, while each leaf node $n\in N$ in the lookup DAG is labeled (*n.target*) with either a user-specified method `m`$_{\texttt{j}}$ or one of the two error methods `m`$_{\texttt{not-understood}}$ and `m`$_{\texttt{ambiguous}}$. Each edge $e=(n_{source},n_{target})\in E$ in the lookup DAG has a source node $e.source=n_{source}$, a target node $e.target=n_{target}$, and is labeled (*e.class*) with a class `Class`$_{\texttt{k}}$. Figure 2 shows a lookup DAG for the running example, where a circle represents an interior node `n` (labeled with `n.expr`), a box represents a leaf node `n` (labeled with `n.target`), an arrow represents a set of edges {`e`$_1$,...,`e`$_k$} all of which have the same source and target nodes (labeled with the various `e`$_i$`.class` values of the edges `e`$_i$), and the entry arrow marks the root node n_0. (The sets subscripting nodes and the dashed circle are used during construction of the lookup DAG, as described later.)

To perform dispatching using a lookup DAG, evaluation follows a path through the DAG, beginning with the root node n_0, in an environment where the names of the generic function's formals are bound to the corresponding actuals for the call being dispatched. To evaluate an interior node *n*, its expression *n.expr* is evaluated to produce a result `v`, the node's outgoing edges *n.edges* are searched for the unique edge *e* whose label *e.class* is the class of `v` (evaluation fails if no such edge is found), and the selected edge's target node *e.target* is evaluated recursively. To evaluate a leaf node *n*, its method *n.target* is returned.

Assumed class hierarchy:

```
object A;
object B isa A;
object C;
object D isa A, C;
```

Assumed source generic function:

```
gf Fun(f1, f2)
  when f1@A and t := f1.x and t@A and (not t@B) and
       f2.x@C and test(f1.y = f2.y) { ...m1... }
  when f1.x@B and ((f1@B and f2.x@C) or (f1@C and f2@A)) { ...m2... }
  when f1@C and f2@C { ...m3... }
  when f1@C { ...m4... }
```

Assumed static class information for expressions (*StaticClasses*):

f1: $\mathcal{C} - \{D\} = \{A,B,C\}$
f2: $\mathcal{C} = \{A,B,C,D\}$
f1.x: $\mathcal{C} = \{A,B,C,D\}$
f2.x: *Subclasses*(C) = {C,D}
f1.y=f2.y: bool = {true,false}

Canonicalized dispatch function:

```
df Fun(f1, f2)
     (f1@A and f1.x@A and f1.x@!B and (f1.y=f2.y)@true) => m1   {c1}
  or (f1.x@B and f1@B) => m2                                   {c2}
  or (f1.x@B and f1@C and f2@A) => m2                          {c3}
  or (f1@C and f2@C) => m3                                     {c4}
  or (f1@C) => m4                                              {c5}
```

Canonicalized expressions and assumed evaluation costs:

e_1=f1 (**cost**=1)
e_2=f2 (**cost**=1)
e_3=f1.x (**cost**=2)
e_4=f1.y=f2.y (**cost**=3)

Constraints on expression evaluation order:

$e_1 \rightarrow_{Expr} e_3$; $e_3 \rightarrow_{Expr} e_1$; $e_1, e_3 \rightarrow_{Expr} e_4$
$e_1, e_3 \leq_{Expr} e_4$

Figure 1: Example for Lookup DAG Construction, Part 1

3.2 Canonicalization of Dispatching Predicates

The first step in constructing the lookup DAG from a generic function is to canonicalize and simplify the form of the predicates of the methods in the generic function. The following grammar defines this canonical form:

```
DF          ::= df Name(Name1,...,Namek) Case1 or ... or Casep
Case        ::= Conjunction => Method1 ... Methodm
Conjunction ::= Atom1 and ... and Atomq
Atom        ::= Expr@Class | Expr@!Class
```

In essence, this grammar represents all the predicates of all the methods of a generic function in disjunctive normal form, i.e., a disjunction of conjunctions, whose atomic predicates are class tests (`Expr@Class`) and negated class tests (`Expr@!Class`). To represent the connection between the original predicates and their associated methods, each conjunction in the canonical form maps to a set of one or more methods, `Conjunction => Method1 ... Methodm`.

Our algorithm converts a regular generic function `GF` into a canonical dispatch function `DF` in the following steps:[1]

Constructed Lookup DAG:

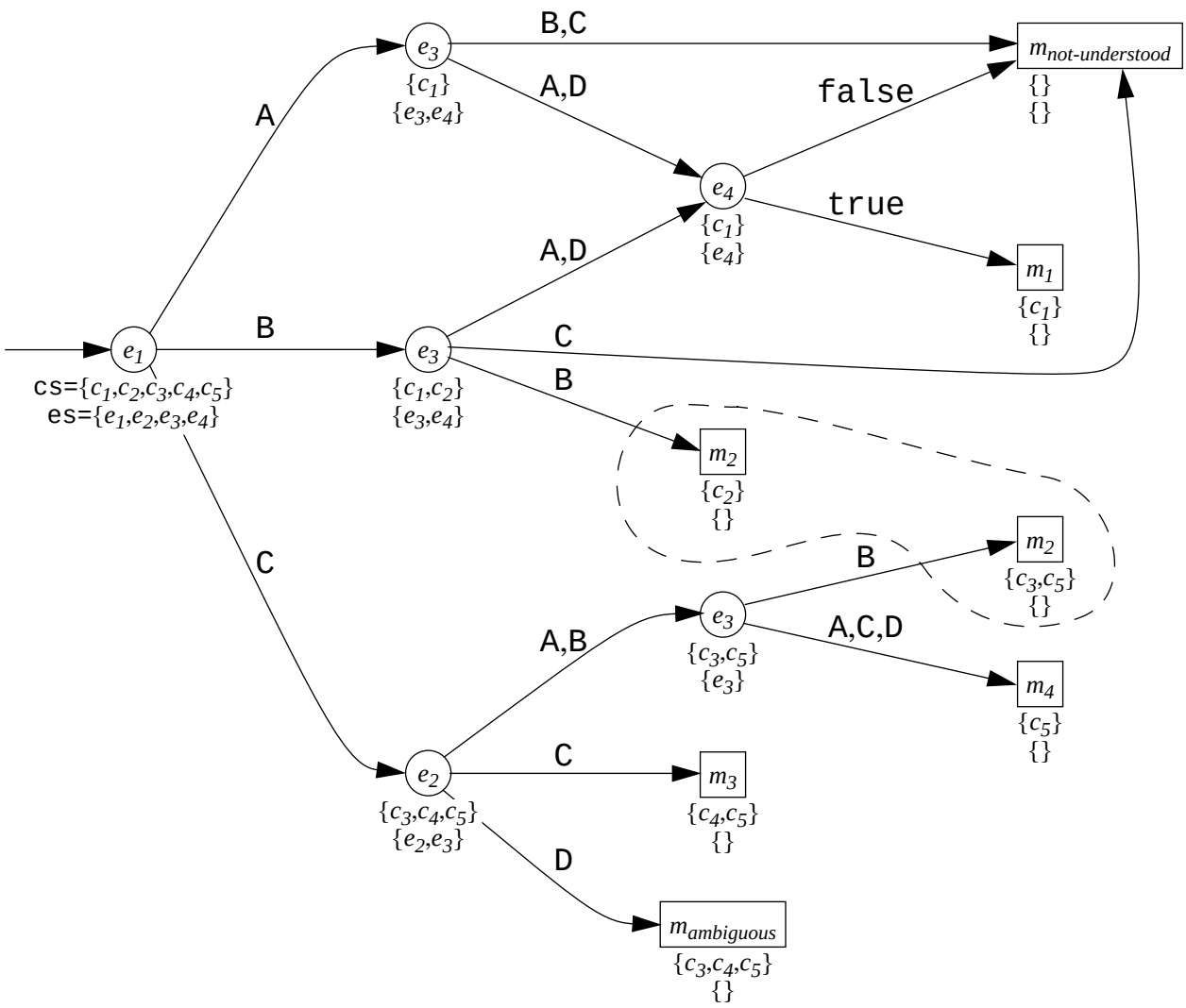

Figure 2: Example for Lookup DAG Construction, Part 2

1. Replace all **`test`** `Expr` clauses with `Expr@True` clauses, where `True` is the (possibly artificial) class of the `true` object. (In a prototype-based language, this test simply becomes `Expr@true`.)

2. Remove all `Name := Expr` clauses and replace references to `Name` in later `Expr'` expressions with `Expr`. (This replacement can be done by sharing the single `Expr` parse tree in all referencing `Expr'` trees, so this rewriting does not increase the size of the predicate expressions.)

3. Convert each method's predicate into disjunctive normal form, i.e., a disjunction of conjunctions of possibly negated `Expr@Class` atomic clauses, using a standard algorithm. (It is possible for this conversion to grow the size of the predicates exponentially, but we do not expect this in practice, nor can it happen for the restricted case of single or multiple dispatching and predicate classes.)

4. Replace **`not`**`(Expr@Class)` clauses with `Expr@!Class` clauses.

[1] The original generic function `GF` has already been canonicalized somewhat by renaming formals so that all methods in the generic function have the same formal names.

5. Place each method `m`'s predicate into canonical form by replacing each conjunction `Conjunction` with `Conjunction => m`.
6. Form the disjunction of all the individual methods' canonicalized predicates, flattening out any resulting disjunctions of disjunctions to recover disjunctive normal form.

Our algorithm exploits available static information about the possible classes of expressions to reduce the size of the canonicalized dispatching predicate. Available static information is represented by a function $\mathcal{StaticClasses}: \mathcal{E} \rightarrow 2^{\mathcal{C}}$ that represents for each expression $e \in \mathcal{E}$ (where $\mathcal{E}$ is the set of expressions occurring in the predicates of the methods in the generic function) the set of possible classes (a subset of $\mathcal{C}$, the set of all classes in the program) of which the result of evaluating e may be a direct instance. *StaticClasses* can be derived from static type declarations (if in a statically typed language) and/or from automatic static class analysis. For a dynamically typed language with no static analysis, a trivial *StaticClasses* function that maps each expression to the set of all classes $\mathcal{C}$ can be used. The following canonicalization steps exploit static class information:

7. Remove all atomic tests that are guaranteed to be true by static class information. In particular, remove atoms of the form `Expr@Class` where $\mathcal{StaticClasses}$(`Expr`) $\subseteq \mathcal{Subclasses}$(`Class`), the set of all subclasses of `Class` (including `Class` itself). Similarly, remove all atoms of the form `Expr@!Class` where $\mathcal{StaticClasses}$(`Expr`) $\cap \mathcal{Subclasses}$(`Class`) $= \emptyset$.
8. Remove all conjunctions containing atomic tests that are guaranteed to be false by static class information. In particular, remove all conjunctions containing atoms of the form `Expr@Class` where $\mathcal{StaticClasses}$(`Expr`) $\cap \mathcal{Subclasses}$(`Class`) $= \emptyset$. Similarly, remove all conjunctions containing atoms of the form `Expr@!Class` where $\mathcal{StaticClasses}$(`Expr`) $\subseteq \mathcal{Subclasses}$(`Class`).

A final clean-up step merges any duplicate conjunctions:

9. Replace the set of all cases of the form $\texttt{Conjunction}_i$ `=>` $\texttt{m}_{ij}$ having the same $\texttt{Conjunction}_i$ with the single case $\texttt{Conjunction}_i$ `=>` $\texttt{m}_{i1}$ `...` $\texttt{m}_{in}$.

Figure 1 gives an example input inheritance graph, generic function, and *StaticClasses* mapping, and shows the canonical dispatch function derived from this generic function. As part of this canonicalization process, the binding of `t` is eliminated by replacing references to it with its value, the negated subclass test is replaced with a single atomic test, the `test` clause is rewritten using a dummy `true` class, the nested disjunction is moved to the outer level (as a result, two different cases now map to the same source method m_2), and the subclass tests on `f2.x` are eliminated as their outcomes are implied by statically known class information.

This canonicalization process in effect reduces the problem of the general predicate dispatching model to the simpler multimethod dispatching model; indeed, canonicalization is largely the identity function on the subset of predicate dispatching corresponding to multimethod dispatching (conjunctions of subclass tests). Each `Case` disjunct in the canonical dispatch tree is analogous to a multimethod, and each `Expr` being tested by the case is analogous to one of the multimethod's formals. Consequently, many of the previous techniques for multimethod dispatching should be applicable to the predicate dispatching model by exploiting this analogy, and conversely our algorithm for constructing dispatch functions for canonicalized dispatch predicates applies to the special case of multimethod dispatching as well.

3.3 Constraints on Expression Evaluation

The second step of our algorithm is to determine minimal constraints on the order of evaluation of the `Expr`s in the canonical dispatch predicate. `Expr`s are not allowed to have externally visible side-effects, and disjuncts can be evaluated in any order or even have their evaluations interleaved, but programmers are allowed to depend on order of evaluation of conjuncts where successful outcomes of earlier conjuncts ensures that later conjuncts do not

function *Cases*(**df** Name**(**Name_1,...,Name_k**)** Case_1 **or** ... **or** Case_p) = {Case_1, ..., Case_p}
function *Methods*(Conjunction **=>** Method_1 ... Method_m) = {Method_1, ..., Method_m}
function *Atoms*(Conjunction **=>** Method_1 ... Method_m) = *Atoms*(Conjunction)
function *Atoms*(Atom_1 **and** ... **and** Atom_q) = {Atom_1, ..., Atom_q}
function *Expr*(Expr@Class) = Expr
function *Expr*(Expr@!Class) = Expr
function *Classes*(Expr@Class) = *Subclasses*(Class)
function *Classes*(Expr@!Class) = $\mathcal{C}$ - *Subclasses*(Class)
function *Subclasses*(Class) = **set of all subclasses of** Class**, including** Class **itself**
value $\mathcal{C}$ = **set of all classes in program**
function *Exprs*(cs) = $\bigcup_{c \in cs}$ *Exprs*(c)
function *Exprs*(c) = { *Expr*(atom) | atom ∈ *Atoms*(c) }

Figure 3: Helper Functions

encounter errors, as described in section 2. Our algorithm could be conservative and assume that all conjunctions must be evaluated left-to-right, but our algorithm will be able to produce faster, smaller dispatch functions if it can reorder conjuncts.

To represent a conservative approximation to the minimal constraints on evaluation order that the lookup DAG must respect, our algorithm computes a partial order $\leq_{\text{Expr}}$ over $\mathcal{E}$, the set of all Exprs in the canonical dispatch predicate; $e_1 \leq_{\text{Expr}} e_2$ if e_1 may need to be evaluated before e_2 to ensure that e_2 does not fail unexpectedly. This partial order is built in two steps. First, a preorder[1] $\rightarrow_{\text{Expr}}$ over $\mathcal{E}$ is defined as the reflexive, transitive closure of basic orderings $e_1 \rightarrow_{\text{Expr}} e_2$ created if e_1 and e_2 both appear in some conjunction, e_1 appears before e_2 in that conjunction, and the compiler cannot prove that evaluation of e_2 would not encounter an error even if the test of e_1 is false (for example, simple references to formals are known statically to be evaluable in any order). A cycle can be formed if e_1 is evaluated before e_2 in one conjunction, but e_2 is evaluated before e_1 in some other conjunction.

The partial order $\leq_{\text{Expr}}$ is derived from the preorder $\rightarrow_{\text{Expr}}$ by ignoring any non-reflexive cyclic orderings: $e_1 \leq_{\text{Expr}} e_2$ iff $e_1 = e_2$ or $e_1 \rightarrow_{\text{Expr}} e_2$ but not $e_2 \rightarrow_{\text{Expr}} e_1$. It is safe to ignore cycles because the orderings computed in $\rightarrow_{\text{Expr}}$ are only conservative approximations to the real constraints, and if in different conjunctions expressions e_1 and e_2 are evaluated in different orders, it must be safe to evaluate them in either order, and hence the cyclic ordering constraints must be unnecessary.

Figure 1 shows the initial preorder and final partial order over expression evaluation for the running example. The preorder is derived from the order of expression evaluation in the various cases of the canonicalized dispatch function, ignoring orderings between tests of independent formals. Because f1's class is tested *before* f1.x is evaluated in one case, but *afterwards* in another case, they can legally be evaluated in either order (semantically, disjuncts can be evaluated in any order), and the final partial order reflects ignoring the cyclic constraints in the initial preorder.

3.4 Construction of Lookup DAG

Our algorithm for constructing the lookup DAG for a canonicalized dispatching function DF is given in Figures 3 and 4. The heart of the algorithm is the function buildSubDag which builds a node and all its subnodes, given a set of candidate cases (cs) and a set of expressions in the candidate cases remaining to be evaluated (es). The root

[1] A preorder is a reflexive, transitive, but not necessarily antisymmetric binary relation. A partial order is an antisymmetric preorder.

```
function buildLookupDag(DF) =
  create empty lookup DAG G
  create empty table Memo
  cs := Cases(DF)
  G.root := buildSubDag(cs, Exprs(cs))
  return G

function buildSubDag(cs,es) =
  if (cs,es)→n ∈ Memo then return n
  if es=∅ then
    n := create leaf node in G
    n.target := computeTarget(cs)
  else
    n := create interior node in G
    expr := pickExpr(es, cs)
    n.expr := expr
    for each class ∈ StaticClasses(expr) do
      cs' := targetCases(cs, expr, class)
      es' := (es - {expr}) ∩ Exprs(cs')
      n' := buildSubDag(cs', es')
      e := create edge from n to n' in G
      e.class := class
    end for
  end if
  add (cs,es)→n to Memo
  return n

function computeTarget(cs) =
  methods := min≤Method( ⋃ c ∈ cs Methods(c))
  if |methods| = 0 then return m_not-understood
  if |methods| > 1 then return m_ambiguous
  return single element m of methods

function pickExpr(es, cs) =
  legal_es := min≤Expr(es)
  discriminating_es := { expr ∈ legal_es | avgNumTargetCases(cs, expr) is minimal }
  cheap_es := { expr ∈ discriminating_es | expr has minimal evaluation cost }
  return any member expr of cheap_es

function avgNumTargetCases(cs, expr) =
  ( Σ c ∈ cs |testedClasses(c, expr)| ) / |StaticClasses(expr)|

function targetCases(cs, expr, class) = { c ∈ cs | class ∈ testedClasses(c, expr) }

function testedClasses(c, expr) =
  if expr ∈ Exprs(c) then return ⋂ atom ∈ Atoms(c), Expr(atom) = expr Classes(atom)
  else return C
```

Figure 4: Lookup DAG Construction Algorithm

node of the DAG corresponds to the set of all cases in `DF` and the set of all expressions in these cases. The table `Memo` memoizes calls to `buildSubDag` so that at most one node for a given set of cases and remaining expressions will ever be constructed, with that node and its successors shared by all earlier nodes leading to it.

When building a new node for a particular set of cases and remaining expressions, if the set of remaining expressions is empty, then the outcome of method lookup has been determined. A leaf node is constructed, and its target (or error) method computed (using `computeTarget`) from the most specific method(s) (under the method overriding partial order $\leq_{\texttt{Method}}$) in the set of methods associated with the cases that reach the leaf. If, on the other hand, the set of remaining expressions is not empty, then more dispatching is necessary. An interior node is constructed, and one of the remaining expressions selected (via `pickExpr`) as the next to evaluate at this interior node. For each of the possible classes resulting from this evaluation, the subset of the input cases is computed whose atomic tests on the selected expression all succeed for that class (by `targetCases`), the set of expressions for that subset is computed by restricting the set of unevaluated remaining expressions to those mentioned in the computed subset of cases, a sub-DAG for the reduced set of cases and remaining expressions is constructed recursively, and finally an edge is created to connect the interior node to the target sub-DAG.

The `pickExpr` function is responsible for selecting the next expression to evaluate from the set of remaining expressions. The one correctness constraint on `pickExpr`'s behavior is that it can only pick expressions that are not required to follow (according to the $\leq_{\texttt{Expr}}$ partial order constructed as described in section 3.3) any other expressions still remaining to be evaluated; these allowed choices are collected into `legal_es`. Within this constraint, we wish to select an expression that will minimize the expected time to proceed through the rest of the lookup DAG. As a heuristic approximation, we pick an expression whose average number of remaining cases in successor nodes is minimized (`discriminating_es`); nodes with no remaining cases are leaves, and nodes with fewer remaining cases are more likely to have shorter paths to leaves than nodes with more remaining cases. If ties still remain, we select an expression that is cheapest to evaluate, using some static estimate of the cost of expression evaluation (`cheap_es`). `pickExpr` is the one place in our high-level algorithm that uses heuristics, and alternative heuristics could be substituted for these without affecting the correctness of our algorithm.

Figure 2 shows the lookup DAG constructed by our algorithm for the running example; the two sets below each node correspond to the (`cs`,`es`) pair mapping to that node in the `Memo` table. At the start of the example, expression e_1 is picked as the first to evaluate, as it is the most discriminating of the legal expressions $\{e_1,e_2,e_2\}$ (e_4 is not legal as it must follow evaluation of other unevaluated expressions). Each of the possible classes for e_1 considered, computing the subset of cases that apply to that outcome (`cs'`) and the set of expressions remaining to be evaluated to distinguish those applicable cases (`es'`). For this example, all three possible classes lead to distinct nodes, which are then processed recursively and independently. Two of the nodes choose to evaluate expression e_3 next, while the third chooses to evaluate e_2. During subsequent recursive construction, the node labeled ($\{c_1\}$,$\{e_4\}$) is analyzed twice, and the sub-DAG starting with that node is shared. When reaching a leaf node, the set of cases reaching that node is used to determine the set of applicable methods for that node, which in turn determines the target or error method to invoke for that leaf. If no applicable methods remain (i.e., if no cases reach that leaf node), the message-not-understood error method is used. If several applicable methods remain but one is more specific than all others (as in the nodes labeled $\{c_3,c_5\}$ and $\{c_4,c_5\}$), then the most-specific method is selected as the target method. If multiple applicable methods remain but none is most-specific (as in the node labeled $\{c_3,c_4,c_5\}$ where m_2 and m_3 are applicable and most-specific), then the message-ambiguous error method is used.

An important feature of our algorithm is that the order of evaluation of expressions can be selected according to heuristics, in `pickExpr`. Some previous work only considered left-to-right evaluation of multimethod arguments [Dussud 89, Chen et al. 94], but alternate orders of evaluation can produce smaller, faster dispatchers. Another

important feature of our algorithm is that each of a node's successor subtrees is computed independently, with different subsets of expressions evaluated along different paths, possibly in different orders, as opposed to previous work which examined the dynamic classes of all formals in all cases. By picking expression evaluation order intelligently in order to reduce average path length, we can produce faster dispatch functions, as occurred in the example in Figure 2.

The complexity of building the lookup DAG, assuming that calls to `testedClasses` are memoized, is $O(D \cdot (C \cdot P + C \cdot E + P \cdot E + M))$, where D is the number of nodes in the constructed lookup DAG, C is the number of concrete classes in the program, P is the size of the predicates (`cs` is $O(P)$), E is the number of expressions in the predicates (`es` is $O(E)$), and M is the number of target methods in the generic function. The number of nodes in the DAG, D, is no worse than exponential in the number of classes in the program C and the number of expressions in the predicate E, and this situation can occur if M or E is also exponential in C, but we do not know if a tighter bound exists on the size of the lookup DAG if we assume that C, P, E, and M are all $O(N)$.[1]

3.5 Postprocessing

Memoization of calls to `buildSubDag` ensures that no two nodes in the lookup DAG have the same set of cases and remaining expressions. However, it is possible that two leaf nodes with different sets of cases still have the same target (or error) method, and these nodes can be safely merged without affecting correctness. After merging all leaf nodes having the same target method, if a merged leaf node's predecessor now has only a single successor node, it can be eliminated. Similarly, if merging leaf nodes into one causes two or more predecessor nodes to have the same successors (and edge labels leading to those successor), the predecessors can be merged. These removing and merging transformations can ripple backwards through the lookup DAG. It is difficult to perform these clean-ups proactively as the lookup DAG is being constructed because it is hard to predict for two different sets of cases and remaining expressions whether the sub-DAGs constructed from them will be isomorphic and lead to the same target methods. Our algorithm instead performs these kinds of clean-up transformations in a single backwards pass over the lookup DAG after construction.

In the lookup DAG shown in Figure 2, there are two leaf nodes (circled with a dotted line) that are labeled with different cases but that both invoke method m_2. Postprocessing would merge these two nodes into a single node. For this example, no additional nodes can be merged.

3.6 Factoring Common Subexpressions

The final step in constructing an efficient lookup DAG is to factor common subexpressions in the expressions being evaluated in the lookup DAG. Common subexpressions across expressions may have been created as a result of eliminating `Name := Expr` clauses during initial canonicalize of predicates as described in section 3.2. They may also have appeared naturally in the original predicate expressions; since predicate expressions are written separately for separate methods, there may not have been any obvious source-level common subexpressions within individual method declarations, but rather subexpressions may have been common only across method declarations.

Our algorithm applies a variant of partial redundancy elimination (a well-studied traditional compiler technique [Morel & Renvoise 79]) to identify common subexpressions and to place computations of those subexpressions in the lookup DAG (possibly creating new intermediate nodes in the graph whose sole purpose is to evaluate some subexpression). Our variant of partial redundancy elimination can ensure that each subexpression is evaluated at most once along any path through the DAG. In effect, this transformation reinserts a minimal number of `Temp :=`

[1] E could even be treated as bounded by a small constant, as is commonly assumed about the number of arguments to a function.

`SubExpr` clauses into the lookup DAG for any common subexpressions. (To save space in this paper, we do not describe this final step in our algorithm further.)

3.7 Comparison with Previous Work

Only a few approaches have previously been devised for efficient multimethod dispatching. Kiczales and Rodriguez [Kiczales & Rodriguez 90] describe a strategy where each generic function has a corresponding hash table, mapping from the tuple of the dynamic classes of the specialized message arguments to the target multimethod. Dussud [Dussud 89] describes a strategy for TICLOS using a tree of hash tables per generic function, each hash table mapping the dynamic class of a particular argument to a nested subtree of hash tables that collectively test the remaining arguments, with leaf tables mapping to the target method. Dussud's tree of single-dispatching hash tables can be viewed as a restricted, less optimized version of our lookup DAG, similar in that it factors out the results of tests of earlier arguments, but different in restricting each lookup DAG node to be implemented with a hash table (our algorithm can customize the implementation of each node separately, as discussed in section 4), producing distinct subtables for each class index in a table (unlike the sharing of successors by multiple class indices of a node and even across nodes), and requiring all dispatched arguments to be evaluated along all paths in a fixed left-to-right order. Kiczales and Rodriguez's algorithm computes a single hash over a tuple of classes rather than a series of hashes over single classes as does our and Dussud's algorithms; a single hash over a tuple may be faster if all arguments need to be tested, but if different methods test different numbers of arguments, then our more selective algorithm may perform better. Both hash-table-based schemes are intended to work on-line, filling the hash tables as the program runs and only recording entries for combinations of classes that occur in the actual program run. The first time a particular combination of argument classes is seen, a more expensive lookup algorithm is performed to fill in the table(s). In contrast, our algorithm computes a single dispatch function for all possible combinations of arguments in one step, in some cases producing a bigger and slower dispatch function than the hash-table-based approaches. We expect our algorithm to be applied off-line at static compile or link time, incurring no run-time cost as a result. The space cost of both of these dynamically filled hash-table-based schemes is proportional to the number of combinations of argument classes used during program execution, which in the worst case is N^k, where N is the number of classes in the program and k is the number of specialized argument positions; the sharing of subtrees in our approach is likely to produce much more compact dispatchers in the common case. Only microbenchmark performance results were reported for these implementations.

Chen and colleagues [Chen et al. 94] developed an approach based on a decision DAG (which they describe as a finite-state automaton) per generic function. Each node in the decision DAG represents a test of a particular argument's dynamic class, with each outgoing edge representing a different outcome; multiple dynamic classes can be represented by a single edge, if they invoke the same set of target methods under the same conditions. By sharing nodes, the space cost of this approach can be much lower than the hash-table-based approaches. Our algorithm was inspired by this earlier work, sharing its underlying DAG-based approach. Our algorithm generalizes Chen's algorithm to support the predicate dispatching model, to test the class of formals in any order (not just left-to-right[1]), to allow testing the classes of different subsets of formals on different paths through the DAG, and to allow the implementation of each node in the DAG to be customized independently (as discussed in section 4). Additionally, Chen's algorithm was not empirically assessed on any benchmarks, while our algorithm is assessed on a collection of large Cecil and Java programs (as described in section 5).

[1] The Chen decision DAG was aimed at Common Lisp-like languages with left-to-right argument prioritization, so a fixed left-to-right evaluation order made sense for the underlying dispatching model. Our predicate dispatching model and lookup DAG algorithm does not directly handle Common Lisp-like dispatching rules.

In general, multimethod dispatching for a generic function of k dispatched arguments can be viewed as indexing into a k-dimensional matrix whose elements are the target (or error) methods to invoke for a lookup, assuming that the class of an object is represented by an integer in the range [0..N-1]. Since this matrix consumes N^k words of space, this approach to dispatching is not directly practical. To reduce the space costs, Amiel *et al.* [Amiel et al. 94] developed techniques for compressing the matrix by finding and merging identical submatrices of the matrix, at the cost of introducing an additional N-long helper array per dispatched argument. The end result is a system that performs k one-dimensional array index operations and one k-dimensional matrix index operation for each dispatch, and consuming $k{\cdot}N+O(N^k)$ space for each generic function. The size of the asymptotically exponential term in the space complexity is critically dependent on the effectiveness of compression. In many situations compression can be good, but for binary methods like `equal` where most classes c define a $c{\times}c$ multimethod case, the diagonal of the matrix is filled with mostly distinct entries, causing most planes to be unique and preventing effective compression. Amiel's algorithm originally was only assessed in terms of space cost, and then only for randomly generated generic functions. Randomly generated generic functions may not resemble real-world generic functions; for example, the all-diagonal matrix is unlikely to be generated randomly, but quite common in practice. Later work [Dujardin et al. 98] used data taken from the Cecil system, but again only space costs were studied.

Dujardin [Dujardin 96] developed an approach using decision DAGs similar to Chen's algorithm and our algorithm. Unlike Chen's algorithm, Dujardin's algorithm targeted unordered multimethods as in Cecil, Dylan, and the multimethod subset of predicate dispatching (not multimethods that prioritize arguments left-to-right as in Common Lisp), which mainly simplifies the algorithm. Dujardin discussed selecting an testing order other than left-to-right, but did not present any specific heuristics. Dujardin's algorithm implements each node's dispatch with two array lookups, using one N-long helper array per dispatched argument and one compressed array per dispatch node, using the same kinds of table compression ideas as in Amiel's algorithm. Dujardin used data from the Cecil system to compare the space cost of his decision DAGs against Amiel's multidimensional matrix-based scheme. His algorithm produces trees with the same space cost as matrices for generic functions specialized on fewer than three arguments, but for generic functions with three or four specialized arguments (the maximum in his data set), dispatch DAGs are roughly half the size of the corresponding compressed matrices. There was no study of the impact on speed of the different dispatching mechanisms.

4 Individual Dispatch Implementation

The lookup DAG constructed by our high-level algorithm leaves unspecified how each interior node is to be implemented. Conceptually, each interior node performs an N-way switch over the N possible classes of the expression being tested by the node. Several different techniques have been used previously to implement this N-way switch:

- In many systems, including most statically typed singly dispatched languages, N-way switches are implemented as lookups in an N-long array, assuming that the class of an object is encoded as an integer in the range [0..N-1].[1] This strategy is most efficient if N is large, most entries of the array are different, and the dynamic frequency distribution over the possible classes is flat.
- In some systems, typically early dynamically typed systems, N-way switches are implemented as dynamically filled hash table lookups [Kiczales & Rodriguez 90, Dussud 89, Kaehler & Krasner 83]. This strategy is efficient if it is hard to predict statically what classes will be used, and the dynamic frequency distribution over the possible classes is flat.

[1] Most singly dispatched languages implement tables stored with each class indexed by an integer identifier of the generic function being invoked. In this paper we consider only the transposed organization: tables stored with the generic function indexed by integer identifiers of the class.

- In some systems, including several more recent dynamically typed systems, *N*-way switches are implemented as linear searches through the possible classes, as in polymorphic inline caches [Hölzle et al. 91]. To make the linear search effective, dynamic profile information is used to order the tests in decreasing order of likelihood, typically by constructing the linear search code dynamically as the program runs. This strategy is efficient if the frequency distribution is highly skewed toward a few classes, as are many call sites in practice. Indeed, linear searching can outperform table-based lookups for highly skewed frequency distributions, since the code for comparing a register against a series of small integer constants is quick while the memory loads and indirect jumps of the table-based schemes can incur expensive pipeline stalls [Driesen et al. 95].
- In some systems, *N*-way switches are implemented as a linear search through subclass tests [Chen et al. 94]. Instead of testing for individual classes, the set of all subclasses of a given class are tested as a unit, where all those subclasses branch to the same target node. Overlapping subclass tests must be performed bottom-up. This approach performs fewer tests in the worst case than linear search through individual classes, but the cost of a single subclass test is more expensive than a single class identity test (e.g., requiring at least an extra load instruction under the various strategies described by Vitek *et al.* [Vitek et al. 97]) and it is difficult to test for commonly occurring classes early if they have (perhaps infrequently occurring) subclasses that branch to different target nodes.
- In a few systems, *N*-way switches are implemented using balanced binary search through the integer encodings of the possible classes [Nakamura et al. 96, Zendra et al. 97]. By taking only logarithmic time rather than linear time, worst-case dispatching is sped up, but previous systems have not exploited profile data to make expected time be better than logarithmic. Space costs can also be reduced over linear search or table lookups, if the number of target nodes is much smaller than the number of possible classes.

None of these techniques dominates all others under all circumstances, and most techniques are the best choice for some commonly occurring circumstance. Despite this mixed result, previous systems have picked a single dispatching mechanism and applied it universally to all dispatches in the system.

Our algorithm instead crafts an *N*-way dispatching code sequence for each node in the lookup DAG individually. We assume that an object represents its class using an integer value, unique to that class, which we call the *class ID*. The code sequence constructed for an interior node first evaluates the node's expression, then loads the result object's class ID, and then performs an *N*-way branch through a combination of equality tests (as in linear searches), less-than tests (as in binary searches), and one-dimensional array lookups, all based on the class ID. Our algorithm attempts to minimize the expected time to perform the *N*-way branch, based on the expected cost in cycles of the different instructions making up the dispatch and the expected frequency distribution of the possible classes, derived either from dynamic profiles [Grove et al. 95] or simple static estimates. Our algorithm also attempts to balance speed against space, avoiding choices that incur great space cost with only minimal speed benefit. By customizing the dispatch implementation to the individual characteristics of each node, we attempt to gain the advantages of most of the above approaches, often in combination, without suffering their disadvantages. This approach applies to any dispatch, including traditional single dispatching.

Hyafil and Rivest have shown than constructing an optimal decision tree with arbitrary tests is NP-complete [Hyafil & Rivest 76]. Hu and Tucker have an O(N log N) algorithm for constructing optimal decision trees using only less-than tests [Hu & Tucker 71], and optimal decision trees using only equality tests or array lookups are easy to construct in linear time. Our problem mixing threes kinds of restricted tests lies in between these extremes; we do not know the whether our problem admits a polynomial-time optimal algorithm.

4.1 Class IDs and Frequencies

We assume that each class has a single associated integer class ID, unique over all classes, stored in each object that is an instance of the class. Our algorithm works correctly for any such assignment, but it will work better if the class IDs of all subclasses of a class are contiguous. If they are, then a pair of comparison operations can be used to implement a subclass test cheaply without additional memory operations. For systems with single inheritance, it is possible to arrange for this contiguity, but for multiple inheritance it is not always possible. In our implementation in Vortex, we simply assign class IDs to classes in a preorder top-down traversal of the inheritance graph; for tree-structured subgraphs of the inheritance graph, class IDs are assigned contiguously, but in the parts of the graph with multiple inheritance, the set of subclasses of a node may have class IDs interleaved with the class IDs of subclasses of other nodes. Our algorithm will simply produce somewhat larger binary trees if this is the case.

Our algorithm is parameterized by a frequency distribution *Frequency*:`ClassID`→`Number` mapping each class ID to its expected frequency. The distribution can be derived from dynamic profile data or from some static estimate. In the absence of dynamic profile data, our implementation uses a static estimate that treats all class IDs leading to non-error states as equally likely, and class IDs guaranteed to lead to error states as 1000 times less likely than non-error IDs.

Figure 5 gives an example of a sequence of class IDs, the target lookup DAG nodes (identified by number) for each class ID, and the relative frequency of each class ID. We will use this as a running example through the rest of this section.

4.2 Dispatch Tree Construction

The interior node's *N*-way branch maps each of the *N* possible class IDs (the labels on the node's outgoing edges) to a target node. Our dispatch tree construction algorithm conceptually partitions this mapping into a set of *intervals*. An interval is a maximal subrange of integer class identifiers [*lo*..*hi*] such that all IDs in the interval map to the same target state or are undefined (i.e., are not in the set of possible classes for this interior node). The intervals for the example class IDs and targets are shown at the bottom of Figure 5.

4.2.1 Binary Tree Construction

The first step of our algorithm, shown in Figure 6, constructs a binary tree over intervals, guided by the expected frequency distribution, in an effort to minimize expected execution time for traversing the binary tree. The resulting binary tree is either a leaf node containing a target lookup DAG node, or an internal node containing a test and two subtrees. The test in an internal node is either an equality or less-than test against a fixed class ID.

The heart of the algorithm, `buildDispatchSubTree`, operates recursively, given a mapping from class IDs to target lookup DAG nodes (`map`), a list of the class IDs sorted in decreasing order of frequency (`sorted_ids`), and the total frequency of the class IDs in the mapping (`frequency`). The algorithm checks for three possible cases at each recursive step:

- If all class IDs map to the same target node, i.e., the map is a single interval, then a leaf node is returned.
- If the most frequent class ID has a relative frequency above some threshold (*Threshold*, set to 40% in our current implementation), then an equality test against this ID is constructed, with a subtree constructed recursively for the remaining IDs if the equality test fails.
- Otherwise, the class ID starting an interval is selected (by `pickDivider`) that most closely divides execution frequency in half [Knuth 68]. A less-than test against this ID is constructed, with the left and right subtrees constructed recursively from the class IDs less-than and greater-than-or-equal-to this ID, respectively. To make the `pickDivider` calculations efficient, our implementation maintains an array of intervals for the class IDs

Class IDs, Targets, and Frequencies:

Class ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Target	1	2	3	1	4	2	5	6	1	7	8	8	8	8	8	8	8	9	9	9	9	8	8	8	8
Freq.	6	7	7	6	8	8	7	6	8	7	10	11	10	9	500	9	10	10	8	15	17	7	8	15	10

Class ID	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
Target	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	11	11	11	11	11	11	11	11	11
Freq.	1	2	2	1	2	2	100	1	2	2	2	1	1	1	2	1	3	3	3	3	3	3	3	3	3

Dispatch Tree and Intervals:

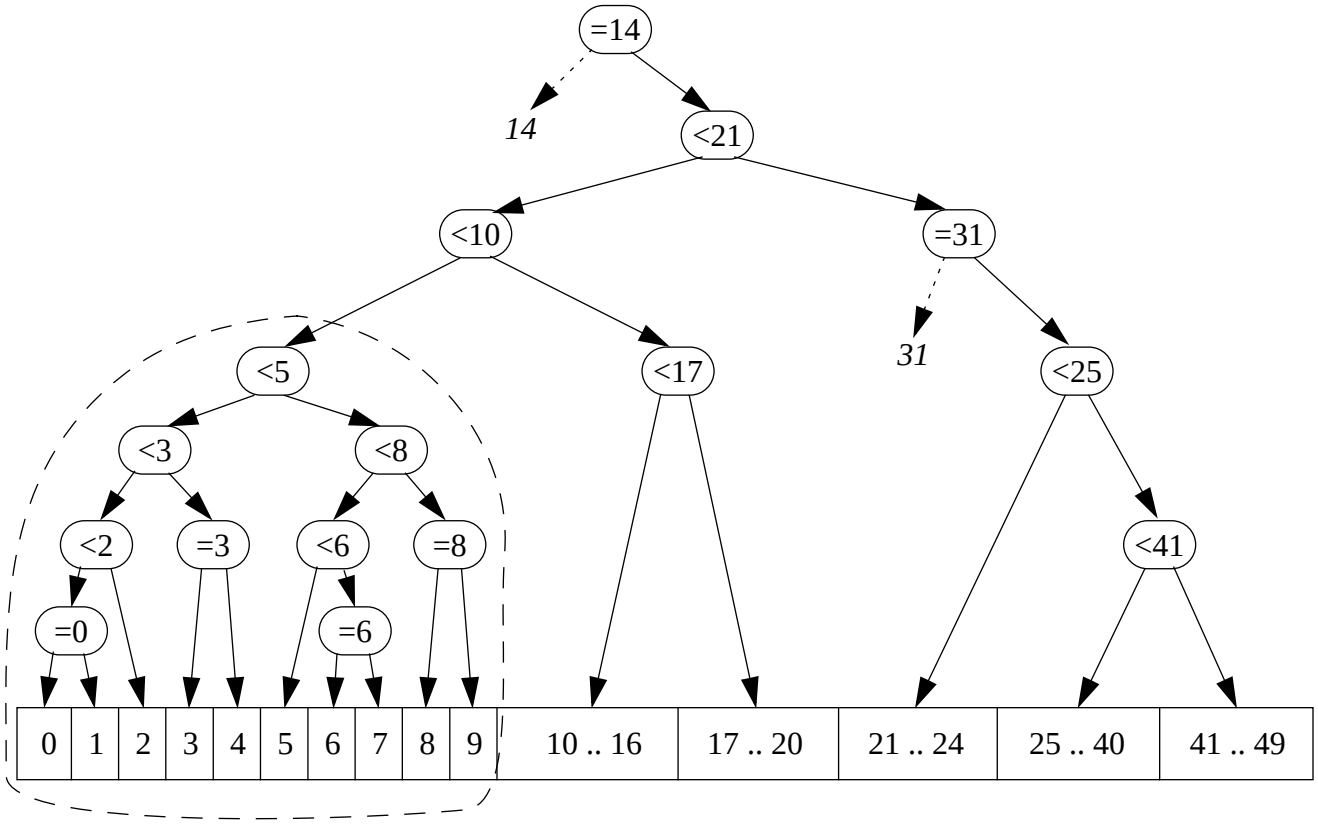

Figure 5: Example for Dispatch Tree Construction

in the domain of `map`, with each interval recording the prefix sum of the execution frequencies of all IDs less than or equal to its high ID.[1]

This algorithm attempts to blend the strengths of linear searching and binary searching. Linear searching works well if the execution profile is strongly peaked, while binary searching works well when the profile is flat. When doing binary searching, by dividing the execution frequencies nearly in half, we hope to balance the weighted expected

[1] As an additional case, our implementation also checks for the case where all but one class ID maps to a given target, and the oddball class ID breaks up the other IDs into two disconnected pieces. In this situation, an equality test for the oddball class ID is constructed, selecting between two leaf nodes. If this special case were not included, and the oddball class ID wasn't very frequent, then two less-than tests would be needed to isolate the three intervals.

```
function buildDispatchTree(n) =
  map := { id→target | e ∈ n.edges ∧ ClassID(e.class) = id ∧ e.target = target }
  sorted_ids := list of all id ∈ dom(map), sorted in decreasing order of Frequency(id)
  frequency := Σ_{id ∈ dom(map)} Frequency(id)
  return buildDispatchSubTree(map, sorted_ids, frequency)

function buildDispatchSubTree(map, sorted_ids, frequency) =
  if |range(map)| = 1 then
    return new leaf(target) where target is single element of range(map)
  best_id := first id ∈ sorted_ids
  if Frequency(best_id) > frequency * Threshold then
    best_target := map(best_id)
    true_subtree := new leaf(map(best_id))
    false_subtree := buildDispatchSubTree(map - {best_id→best_target},
                                          sorted_ids - best_id,
                                          frequency - Frequency(best_id))
    return new node(new test(=,best_id), true_subtree, false_subtree)
  divide_id, frequency_below, frequency_above := pickDivider(map)
  true_subtree  := buildDispatchSubTree(map whose domain is restricted to those < divide_id,
                                        sorted_ids restricted to those < divide_id,
                                        frequency_below)
  false_subtree := buildDispatchSubTree(map whose domain is restricted to those ≥ divide_id,
                                        sorted_ids restricted to those ≥ divide_id,
                                        frequency_above)
  return new node(new test(<,divide_id), true_subtree, false_subtree)

function pickDivider(map) =
  return id ∈ dom(map), frequency_below, frequency_above such that
    id' ∈ dom(map) is maximum such that id' < id,
    map(id) != map(id'),
    frequency_below := Σ_{id ∈ dom(map), id < divide_id} Frequency(id)
    frequency_above := Σ_{id ∈ dom(map), id ≥ divide_id} Frequency(id)
    |frequency_below - frequency_above| is minimal
```

Figure 6: Dispatch Tree Construction Algorithm

execution time of traversing the two subtrees and thereby minimize the total expected execution time for the whole tree.

The time to construct the binary decision tree using our algorithm is $O(C^2)$, where C is the number of possible classes of the expression being tested.

Figure 5 shows the results of this phase of the algorithm on the running example. In the first call to buildDispatchSubTree, class ID 14 has more than 40% of the total frequency, and so an equality test is created for it, and the remaining IDs processed recursively. In the remaining IDs, no single ID has more than 40% of the total frequency, and so the interval boundary that most evenly divides the remaining execution frequency is identified (21), a less-than test created at this ID, and the two subranges processed recursively. Within the right subrange, ID 31 now has more than 40% of the frequency within that subrange, and so another equality test is inserted. Binary tree construction proceeds in this manner, until ranges made up of a single interval are encountered and represented by leaves in the binary tree.

4.2.2 Lookup Array Construction

After constructing the binary tree, our algorithm attempts to find subtrees in the binary tree that are more profitably represented by array lookups. Any subtree can be replaced by an array lookup leaf node by allocating an array with index domain equal to `[lo..hi]` where `lo` and `hi` are the smallest and greatest class IDs handled by that subtree, filling in the defined array elements with the corresponding target lookup DAG node. (In our implementation, if the target lookup DAG node is itself a leaf DAG node `n` holding a target method `n.target`, we store the address of `n.target` in the array element.)

Our algorithm works bottom-up over the binary tree, computing for each interior node in the tree the expected traversal time and actual space costs for that subtree when represented as a binary tree and when represented as an array. The time and space costs are computed based on constants specifying the time and space costs for the various machine instructions and data space using in the implementation of the subtree, with the frequency distribution being used to weight different paths through the decision tree. For each node, we make a decision between implementing the decision as a binary tree or as an array, comparing the expected speedup against the expected space increase for switching to an array. Our current implementation uses the following comparison, switching to an array-based implementation if the comparison is true:

$$\left(\frac{t_{tree}}{t_{array}} - 1\right) \cdot \mathit{frequency} > \left(\frac{s_{array}}{s_{tree}} - 1\right) \cdot \mathit{space\text{-}speed\text{-}tradeoff}$$

In this equation, t_{tree} and t_{array} are the expected time to traverse the tree and array, respectively, and s_{tree} and s_{array} are the cost in space to represent the tree and array, respectively. *frequency* represents the sum of the frequencies of the class IDs tested by the subtree relative to the sum of the frequencies of all class IDs, and it weights the expected time overhead of trees vs. arrays. *space-speed-tradeoff* is used to convert overhead in terms of space into equivalent units of time overhead, representing the balance between the importance of saving time vs. the importance of saving space. In our implementation, *space-speed-tradeoff* is 0.15, indicating that an incremental increase in space cost of a factor of X is matched by an incremental increase in speed of a factor of $0.15X$; speed benefits are nearly 7 times more important than space costs in our system.

The results of this second phase of the tree-construction algorithm are shown in Figure 5, where the largest subtree replaced by a single array lookup is circled, for some assumptions about the relative time and space costs of equality and less-than tests versus array lookups.

This bottom-up postpass over the binary tree constructed by the previous phase, selectively replacing subtrees by arrays, has the advantage of always improving the implementation of the dispatch tree (under assumptions of correctness of the time and space estimates) and taking time linear in the size of the original binary tree. However, it only considers those few subranges of class IDs that correspond to subtrees, and many subranges that could make better arrays are not considered. Future work includes studying algorithms that better integrate selection of good array subranges with decision tree construction.

4.3 Comparison with Previous Work

We are not aware of any previous work that blends any combination of linear searching, binary searching, and array lookups in a single dispatching strategy. All previous work studied a single strategy, applied universally across all dispatch sites.

Some previous work has used subclass tests in dispatching [Chen et al. 94]. Our algorithm does not use subclass tests, but instead uses combinations of less-than tests on class IDs to approximate combinations of subclass tests. Subclass tests appear to require linear time to dispatch over a set of N subclass tests, while less-than tests can require only logarithmic time over the set of M intervals by exploiting binary search. M is at least as big as N, but in our

practical experience, the ratio of *M* to *N* averages less than two, even in the presence of complex multiple inheritance hierarchies. Additionally, simple comparison tests are significantly faster than subclass tests, and so binary search over class IDs at least is competitive in performance with and probably much faster than linear search through subclass tests. Finally, our algorithm can mix in class equality tests to speed the handling of commonly occurring classes.

Some previous work dispatches on several multimethod arguments simultaneously, including Amiel *et al.*'s multidimensional matrix lookup scheme [Amiel et al. 94, Dujardin et al. 98] and Kiczales and Rodriguez's hash table scheme [Kiczales & Rodriguez 90], while our algorithm only considers compositions of single dispatches. It would be interesting to see whether there exist cases where it would be profitable to extend our algorithm to support dispatch nodes that branch on the classes of multiple expressions simultaneously.

5 Experimental Assessment

We implemented our algorithm in the context of the Vortex optimizing compiler for object-oriented languages [Dean et al. 96, Chambers et al. 96]. Our algorithm can be invoked to construct dispatchers for generic functions for a variety of static class information and dynamic profile information, but in our current implementation we apply our algorithm to each generic function once to compute a single dispatching function valid for and shared by all possible call sites of that generic function. (Our algorithm could alternatively be used to create multiple dispatch functions for a single generic function, each specialized to different static class information and dynamic profiles and used for a subset of the possible call sites of the generic function.) We also implement the lookup DAGs and dispatch trees directly as executable code.[1] (Alternatively one could build a data structure representing the lookup DAG and dispatch trees and then interpret these data structures at run-time, perhaps more compactly but much more slowly.)

In assessing the effectiveness of our algorithm, we are primarily concerned with how well our algorithm performs in practice on large, real programs. We applied our algorithm to the collection of benchmark programs described in Table 1.

Table 1: Benchmarks

Language	Benchmark	Size (app. lines + library lines)	Description
Cecil	`instr-sched`	2,400 + 10,700	global instruction scheduler
	`typechecker`	20,000 + 10,700	typechecker for old Cecil type system[1]
	`tc2`	23,500 + 10,700	typechecker for new Cecil type system[1]
	`compiler`	50,000 + 10,700	old version of Vortex optimizing compiler
Java	`cassowary`	3,400 + 16,400	incremental constraint solver
	`toba`	3,900 + 16,400	Java-bytecode-to-C translator
	`java-cup`	7,800 + 16,400	parser generator
	`espresso`	13,800 + 16,400	Java compiler[2]
	`javac`	25,500 + 16,400	Java compiler[2]
	`pizza`	27,500 + 16,400	Pizza compiler
	`javadoc`	29,000 + 16,400	documentation generator

[1] We have implemented translations from the constructed lookup DAGs directly into C or SPARC assembly code and indirectly through Vortex intermediate code. This last option supports applying Vortex's optimizations to the dispatch functions themselves, enabling for example inlining of short callee methods into the dispatching function itself.

1. The two typecheckers are separate pieces of code, using different data structures and algorithms, and were written by different people
2. The two Java compilers are separate pieces of code, using different data structures and algorithms, and were written by different people.

The Cecil programs make heavy use of multiple dispatching and light use of predicate classes; Cecil is the language supporting the largest subset of the predicate dispatching model in which large benchmark programs are available. The Java programs use only single dispatching, but the dispatch tree implementation techniques of section 4 apply to singly dispatched languages as well.

5.1 Lookup DAG Results

Our first experiment studies the structure of lookup DAGs produced for the generic functions in Cecil and Java programs. We examine only the two largest Cecil and Java programs, `compiler` and `javadoc`; the smaller programs produced simpler lookup DAGs. For each generic function, we computed the number of methods in the generic function, the number of tested expressions (dispatched formals and predicate class test expressions) in the generic function, the number of interior nodes, the number of edges (merging edges with the same source and target node), and the average (not weighted by frequency) and maximum path length through the lookup DAG to a non-error leaf node; where sensible, we also computed the total of these values across all generic functions. We simulated two versions of each program: one where no static class information was available (all type declarations are ignored, and all classes are assumed possible at all tested expressions, except for predicate class test expressions which are known to return a boolean) and one where static class information is assumed to be sufficient to ensure that no message-not-understood errors can occur.

The results for the Cecil benchmark are shown in Table 2. This program has 861 concrete classes (57% of which had multiple inheritance somewhere in their ancestry) and 6,860 generic functions. Most generic functions are small, with a couple of methods and typically a single dispatched expression. But some generic functions are quite large (5 generic functions have more than 100 methods in them) and some generic functions have several dispatched expressions (312 generic functions have two tested expressions, 63 have three, 11 have four, and three have five tested expressions). The number of interior states and edges in the constructed DAGs correlates with the structure of the generic functions; the statically typed versions for small generic functions are simplified by ruling out the message-not-understood error case. For the more complex generic functions, the average path length is below the maximum path length, showing benefits for not dispatching on all tested expressions.

Table 2: Lookup DAG Measurements (Cecil)

`compiler`		minimum	median	average	maximum	total
# of methods per generic function		0	1	1.8	542	12,412
# of expressions per generic function		0	1	0.96	5	6,556
dynamically typed	# of interior nodes	0	1	1.1	172	7,299
	# of edges	0	2	3.0	793	20,555
	avg. path length	0	1	0.95	4.0	
	max. path length	0	1	0.96	5	
statically typed	# of interior nodes	0	0	0.33	167	2,248
	# of edges	0	0	1.3	616	8,713
	avg. path length	0	0	0.22	3.0	
	max. path length	0	0	0.23	5	

The results for the Java benchmark are shown in Table 3. This program has 271 concrete classes (57% of which had multiple inheritance or subtyping from interfaces somewhere in their ancestry) and 3,916 generic functions. From this data, we conclude that this largest Java program is much less complicated and less object-oriented than the largest Cecil program; the statically typed version has only 190 generic functions requiring dynamic dispatching in the whole program.

Table 3: Lookup DAG Measurements (Java)

`javadoc`		minimum	median	average	maximum	total
# of methods per generic function		1	1	1.2	35	3,916
# of expressions per generic function		0	1	0.73	1	2,875
dynamically typed	# of interior nodes	0	1	0.73	1	2,875
	# of edges	0	2	1.7	36	6,535
	avg. path length	0	1	0.71	1	
	max. path length	0	1	0.71	1	
statically typed	# of interior nodes	0	0	0.049	1	190
	# of edges	0	0	0.26	35	998
	avg. path length	0	0	0.024	1	
	max. path length	0	0	0.024	1	

5.2 Dispatch Tree Results

Our second experiment studies the structure of the dispatch trees produced for the interior nodes in the lookup DAGs for the Cecil and Java programs. Again, we examine only the two largest Cecil and Java programs, `compiler` and `javadoc`. To compare the effectiveness of our algorithm mixing array lookups, equal tests, and less-than tests (named `=`, `<`, `[]` in tables) against simpler, less flexible algorithms, we also implemented four restricted versions of our algorithm, using only array lookups (named `[]`), only equal tests (implementing linear search as in polymorphic inline caches [Hölzle et al. 91]) (named `=`), only less-than tests (implementing binary search as in TWiST [Nakamura et al. 96] and SmallEiffel [Zendra et al. 97]) (named `<`), and only equal and less-than tests (implementing the binary tree subpart of our algorithm from section 4.2.1 without the array replacement subpart from section 4.2.2) (named `=`, `<`). We studied both dynamically and statically typed versions of the Cecil and Java programs. We also studied deriving class frequency distributions from both dynamic profile data and static estimates.

For each dispatch tree to be constructed, we compute the range of class IDs, the number of mapped class IDs in this range (ignoring impossible class IDs), the number of intervals, and ratio of the number of intervals to the number of target nodes mapped by these class IDs. For each of the five construction algorithms (our full algorithm and the four restrictions), we compute the number of test nodes in the constructed dispatch tree (broken down into the number of equal, less-than, and array-lookup test nodes in the tree), the size in bytes of the machine code and data to represent the dispatch tree, and the expected average time cost in cycles for dispatching. We also compute the total number of the different kinds of tests and the total space cost over all dispatch trees in the program.

The results for the Cecil benchmark are shown in Table 4. In the dynamically typed versions, all expressions corresponding to predicate class boolean tests have only 2 possible classes, while all formal expressions have all 861 possible classes. The total space cost for the 7299 decision trees (the vast majority of the overall space cost of the dispatch functions) in the dynamically typed versions is usually near 200KB, but pure linear search consumes over 1MB of space and pure arrays consume over 6MB of space (pure arrays include no compression techniques to reduce the cost of the arrays; in our system, mixing equal and less-than tests with array lookups supports more

Table 4: Dispatch Tree Measurements (Cecil)

compiler			minimum	median	average	maximum	total
dynamically typed	range of class IDs		2			861	
	count of class IDs		2			861	
	# of intervals		2	3	4.8	585	
	intervals per target		1	1.5	1.8	9.5	
	estimated profiles, =, <, []	# of = tests	0	2	2.2	17	16,305
		# of < tests	0	1	1.5	23	10,771
		# of [] tests	0	0	0.046	4	333
		size	2	18	26.	867	192,772
		avg. dispatch time[1]	2.5	10.	9.2	17.	
	estimated profiles, =, <	# of = tests	1	2	2.5	237	17,906
		# of < tests	0	1	1.8	345	12,950
		size	2	18	26.	867	192,772
		avg. dispatch time	2.5	10.	9.2	17.	
	estimated profiles, =	# of = tests	1	3	31.	859	224,545
		size	2	18	167.	867	1,217,230
		avg. dispatch time	2.5	8.0	8.9	29.	
	estimated profiles, <	# of < tests	0	2	3.8	583	27,837
		size	2	18	29.	867	211,820
		avg. dispatch time	2.5	9.8	10.	17.	
	dynamic profiles, =, <, []	# of = tests	0	2	2.5	24	18,218
		# of < tests	0	1	1.4	22	10,151
		# of [] tests	0	0	0.42	4	305
		size	2	18	26.	867	190,054
		avg. dispatch time	2.5	8.5	8.9	17.	
	dynamic profiles, =, <	# of = tests	1	2	2.8	269	20,364
		# of < tests	0	1	1.6	325	11,680
		size	2	18	26.	867	190,054
		avg. dispatch time	2.5	8.5	8.9	17.	
	dynamic profiles, =	# of = tests	1	3	29.	859	213,553
		size	2	18	138.	883	1,008,595
		avg. dispatch time	2.5	8.0	8.7	25.	
	dynamic profiles, <	# of < tests	0	2	3.8	583	27,837
		size	2	18	28.	867	205,610
		avg. dispatch time	2.5	9.8	10.	17.	
	[]	# of [] tests	1				7299
		size	8			867	6,136,676
		avg. dispatch time	8.0				

Table 4: Dispatch Tree Measurements (Cecil)

compiler			minimum	median	average	maximum	total
statically typed	range of class IDs		2	64	254.	861	
	count of class IDs		2	8	126.	861	
	# of intervals		2	3	4.7	585	
	intervals per target		1	1	1.2	8.5	
	estimated profiles	=, <, []	# of = tests	0	1	0.69	5	1,561
			# of < tests	0	1	1.3	16	2,858
			# of [] tests	0	0	0.15	4	347
			size	2	13	20.	867	44,008
			avg. dispatch time	2.5	6.3	6.7	13.	
		=, <	# of = tests	0	1	1.3	237	2,820
			# of < tests	0	1	2.4	345	5,432
			size	2	13	20.	867	44,008
			avg. dispatch time	2.5	6.3	6.7	13.	
		=	# of = tests	0	3	81.	859	182,796
			size	2	13	139.	867	312,721
			avg. dispatch time	2.5	8.0	7.3	22.	
		<	# of < tests	0	2	3.6	583	8,179
			size	2	13	20.	867	45,285
			avg. dispatch time	2.5	6.6	6.8	13.	
	dynamic profiles	=, <, []	# of = tests	0	1	1.4	17	3,084
			# of < tests	0	1	1.2	16	2,616
			# of [] tests	0	0	0.15	5	326
			size	2	13	21.	867	47,152
			avg. dispatch time	2.5	5.8	6.5	12.	
		=, <	# of = tests	0	1	2.3	269	5,094
			# of < tests	0	1	2.0	325	4,429
			size	2	13	21.	867	47,152
			avg. dispatch time	2.5	5.8	6.5	12.	
		=	# of = tests	0	3	76.	859	171,456
			size	2	16	122.	883	274,916
			avg. dispatch time	2.5	7.2	6.9	22.	
		<	# of < tests	0	2	3.6	583	8,179
			size	2	13	19.	867	43,322
			avg. dispatch time	2.5	6.4	6.7	13.	
	[]		# of [] tests	1				2248
			size	8	70	260.	867	584,305
			avg. dispatch time	8.0				

1. Average dispatch time for versions with estimated profiles uses the estimate, not the dynamic profile, to weight the cost of the various paths through the decision tree.

selective use of arrays). The total space cost for the 2248 decision trees in the statically typed versions is usually under 50KB, but again pure linear search and pure arrays perform relatively poorly.

The results for the Java benchmark are shown in Table 5. Java results are similar to Cecil results, but scaled down reflecting the smaller number of dispatch trees in the benchmark program.

5.3 Performance Results

Our final experiment assesses the bottom-line impact of our dispatching algorithm on the execution speed of all of our benchmark programs. To gain a realistic estimate of the impact of our optimizations, we first apply all of Vortex's aggressive static and profile-guided optimizations to try to optimize dynamic dispatches without recourse to the run-time dispatching mechanism; only those dispatches that could not otherwise be optimized invoke our generated dispatch functions. We measure the performance of each of our benchmarks under each of our five construction algorithms, with static and dynamic typing, and with profile-driven and estimated frequency distributions. As a point of rough comparison, we also report the performance of Vortex's existing polymorphic inline cache-based dispatchers (PICs). Vortex's PICs have an advantage over our new generated dispatchers by being constructed for each call site separately, specialized to the classes occurring at run-time at each call site, but they are worse at handling multiple dispatching and predicate classes than our new generated dispatchers.

The results for the Cecil benchmarks are presented in Table 6, as speed-ups relative to the PIC-based implementation.[1] Our dispatching functions yield speed improvements over the previous PIC strategy (despite PIC's advantage in producing call-site-specific, on-line profile-guided dispatchers) of up to 30%, with bigger speed-ups accruing to bigger programs. (The biggest speed-ups are due in part to the PIC strategy's poorer handling of multiple dispatching.) Our flexible dispatching strategy mixing equal, less-than, and array-based lookups generally produces the best speed-ups, although purely binary search using less-than tests works nearly as well. Linear search using equal tests performs poorly without dynamic profile information, as expected, but the other techniques are less dependent on dynamic profile guidance. Purely array-based lookups were worse than the other techniques. Assuming static type checking had some benefit, particularly in the `tc2` benchmark.

The results for the Java benchmarks are presented in Table 7. Some Java programs were relatively unaffected by switching from PICs to our new dispatching functions (`toba`, `java-cup`, `pizza`, `javadoc`), some saw moderate speed-ups (`cassowary`, `espresso`), and one saw large speed-ups of around 40% (`javac`). Unlike the Cecil results, purely array-based lookups performed competitively with other configurations. Also, assumptions of static type checking did not help speed dispatch. Caching conflicts must be responsible for certain unexpected slow-downs, such as the fact that array-based lookups in the dynamically typed model almost always outperformed, sometimes dramatically, those in the statically typed model, even though the only difference in the two versions is that the arrays in the statically typed version are smaller.

6 Conclusions and Future Work

We have developed a new algorithm for building message dispatch functions. Our algorithm handles the general predicate dispatching model, which includes multiple dispatching, predicate classes and similar constructs, and pattern matching as special cases; our algorithm produces dispatch functions for the special cases of multimethods and predicate classes that usually are more efficient than previous algorithms tailored for that special case. We have also developed a new algorithm for implementing a single dispatch that constructs a blend of array lookups, class equality testing, and class ID less-than testing tailored to the characteristics of each dispatch separately. We have

[1] Execution times were measured on UltraSPARC-1/170 workstations with ~200MB RAM, taking the median time of 11 runs of the benchmark. On this hardware, median times still vary by a few percent, so small differences in computed speed-up should not be considered significant.

Table 5: Dispatch Tree Measurements (Java)

javadoc			minimum	median	average	maximum	total
dynamically typed	range of class IDs		271				
	count of class IDs		271				
	# of intervals		1	3	3.4	54	
	intervals per target		1.0	1.5	1.5	2.5	
	estimated profiles, =, <, []	# of = tests	0	1	1.1	7	3,090
		# of < tests	0	0	0.39	10	1,124
		# of [] tests	0	0	0.014	2	39
		size	3	8	13.	277	36,213
		avg. dispatch time	3	5.5	6.2	12.	
	estimated profiles, =, <	# of = tests	0	1	1.2	17	3,344
		# of < tests	0	0	0.55	39	1,593
		size	3	8	13.	277	36,213
		avg. dispatch time	3	5.5	6.2	12.	
	estimated profiles, =	# of = tests	0	1	5.7	270	16,479
		size	3	8	30.	277	86,792
		avg. dispatch time	3	5.5	6.3	15.	
	estimated profiles, <	# of < tests	0	2	2.4	53	7,002
		size	3	13	16.	277	46,939
		avg. dispatch time	3	7.6	7.7	12.	
	dynamic profiles, =, <, []	# of = tests	0	1	1.1	9	3,134
		# of < tests	0	0	.40	10	1,159
		# of [] tests	0	0	0.016	3	47
		size	3	8	12.	277	35,145
		avg. dispatch time	3	5.5	6.2	12.	
	dynamic profiles, =, <	# of = tests	0	1	1.2	17	3,366
		# of < tests	0	0	0.55	39	1,576
		size	3	8	12.	277	35,145
		avg. dispatch time	3	5.5	6.2	12.	
	dynamic profiles, =	# of = tests	0	1	5.7	270	16,479
		size	3	8	30.	287	86,195
		avg. dispatch time	3	5.5	6.3	15.	
	dynamic profiles, <	# of < tests	0	2	2.4	53	7,002
		size	3	13	16.	277	45,964
		avg. dispatch time	3	7.6	7.7	12.	
	[]	# of [] tests	1				2,875
		size	277				796,375
		avg. dispatch time	8.0				

Table 5: Dispatch Tree Measurements (Java)

javadoc			minimum	median	average	maximum	total
statically typed	range of class IDs		2	20	48.	271	
	count of class IDs		2	4	29.	271	
	# of intervals		2	3	6.5	54	
	intervals per target		1.0	1.0	1.2	2.0	
	estimated profiles	=, <, []	# of = tests	0	1	0.56	2	107
			# of < tests	0	0	0.57	4	110
			# of [] tests	0	0	0.29	1	55
			size	8	8	24.	277	4,602
			avg. dispatch time	5.5	5.5	6.5	9.7	
		=, <	# of = tests	0	1	1.9	13	369
			# of < tests	0	1	3.4	39	643
			size	8	8	24.	277	4,602
			avg. dispatch time	5.5	5.5	6.5	9.7	
		=	# of = tests	1	2	23.	270	4,445
			size	8	10	36.	277	6,833
			avg. dispatch time	5.5	7.5	6.9	12.	
		<	# of < tests	1	2	5.5	53	1,049
			size	8	10	25.	277	4,742
			avg. dispatch time	5.5	5.7	6.6	9.7	
	dynamic profiles	=, <, []	# of = tests	0	1	0.69	4	131
			# of < tests	0	0	0.74	9	140
			# of [] tests	0	0	0.32	3	60
			size	8	8	22.	242	4,140
			avg. dispatch time	5.5	5.5	6.4	9.7	
		=, <	# of = tests	0	1	2.0	13	388
			# of < tests	0	1	3.3	39	626
			size	8	8	22.	242	4,140
			avg. dispatch time	5.5	5.5	6.4	9.7	
		=	# of = tests	1	2	23.	270	4,437
			size	8	10	36.	287	6,903
			avg. dispatch time	5.5	7.0	6.8	12.	
		<	# of < tests	1	2	5.5	53	1,049
			size	8	10	23.	277	4,338
			avg. dispatch time	5.5	5.7	6.6	9.7	
	[]		# of [] tests	1				190
			size	8	26	54.	277	10,216
			avg. dispatch time	8				

Table 6: Benchmark Performance Measurements (Cecil)

language	benchmark	version	dynamically typed		statically typed	
			estimated profiles	dynamic profiles	estimated profiles	dynamic profiles
Cecil	instr-sched	=, <, []	1.08	1.09	1.11	1.12
		=, <	1.09	1.10	1.06	1.12
		=	0.86	1.05	0.94	1.12
		<	1.12	1.09	1.09	1.14
		[]1		1.00		1.10
	typechecker	=, <, []	1.12	1.12	1.12	1.14
		=, <	1.11	1.08	1.11	1.13
		=	0.81	0.96	0.81	0.95
		<	1.12	1.11	1.11	1.14
		[]		1.06		1.10
	tc2	=, <, []	1.13	1.16	1.22	1.23
		=, <	1.14	1.17	1.15	1.15
		=	0.85	1.10	0.87	1.12
		<	1.16	1.18	1.14	1.19
		[]		1.08		1.15
	compiler	=, <, []	1.28	1.26	1.32	1.29
		=, <	1.21	1.28	1.30	1.33
		=	0.75	1.26	0.76	1.28
		<	1.26	1.27	1.32	1.31
		[]		1.16		1.25

1. Profile information has no effect on purely array-based lookups.

assessed our work on a collection of large benchmark programs, using a compiler incorporating a suite of static and profile-guided message optimizations, to demonstrate the bottom-line impact of our new algorithms.

Our current experiments have studied the problem of constructing a dispatch function given complete knowledge of each generic function and the program's whole class hierarchy, as would occur in a link-time whole-program optimizer or dynamic compiler. In the future we plan to study constructing dispatchers by quickly splicing together fragments computed from partial views of the program (such as could be done in a separate-compilation-based environment), to study constructing and adapting dispatching functions on-line as the program runs (exploiting on-line rather than off-line knowledge of the program's execution profile), and to study selective specialization of dispatch functions for the more specific characteristics found at particular call sites. We also wish to rewrite Vortex's static optimizations of dynamic dispatches to be guided directly by a lookup DAG constructed for that call site's static argument class information instead of its current *ad hoc* and restricted approach.

Acknowledgments

Michael Ernst in particular but also Craig Kaplan, David Grove, Todd Millstein, Jonathan Nowitz, and Jonathan Aldrich contributed to this work in early discussions about more general and flexible dispatching algorithms. Vassily Litvinov provided useful comments on a draft of this paper. Richard Ladner developed several optimal polynomial-

Table 7: Benchmark Performance Measurements (Java)

language	benchmark	version	dynamically typed		statically typed	
			estimated profiles	dynamic profiles	estimated profiles	dynamic profiles
Java	cassowary	=, <, []	1.05	1.08	1.07	1.09
		=, <	1.08	1.07	1.08	1.08
		=	1.06	1.08	1.05	1.07
		<	1.09	1.08	1.08	1.09
		[]		1.06		1.07
	toba	=, <, []	0.97	1.01	0.94	0.93
		=, <	0.99	0.98	0.97	0.97
		=	0.90	0.93	0.89	0.91
		<	0.98	0.99	0.91	0.99
		[]		0.95		0.93
	java-cup	=, <, []	1.01	1.00	1.07	1.12
		=, <	1.01	1.00	1.01	1.13
		=	0.97	1.02	0.98	0.96
		<	0.97	1.00	1.12	1.07
		[]		1.05		0.95
	espresso	=, <, []	1.11	1.15	1.08	1.08
		=, <	1.14	1.14	1.13	1.12
		=	1.07	1.13	1.09	1.14
		<	1.12	1.11	1.12	1.12
		[]		1.20		1.08
	javac	=, <, []	1.40	1.41	1.33	1.36
		=, <	1.38	1.39	1.33	1.33
		=	1.11	1.30	1.10	1.34
		<	1.26	1.26	1.28	1.28
		[]		1.45		1.35
	pizza	=, <, []	1.05	1.04	1.03	0.95
		=, <	1.03	1.05	0.98	1.00
		=	0.99	1.01	0.98	1.04
		<	1.04	1.04	1.02	1.03
		[]		1.07		1.03
	javadoc	=, <, []	1.00	1.03	1.01	1.03
		=, <	1.01	1.01	1.01	1.03
		=	0.97	1.00	0.98	1.01
		<	1.01	1.01	1.01	1.01
		[]		1.20		1.01

time algorithms for restricted cases of the dispatch tree construction problem. Jonathan Bachrach and Wade Holst discussed with us general issues and techniques for multimethod dispatching.

This research is supported in part by an NSF grant (number CCR-9503741), an NSF Young Investigator Award (number CCR-9457767), and gifts from Sun Microsystems, IBM, Xerox PARC, Object Technology International, Edison Design Group, and Pure Software.

References

[Agesen 95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[Amiel et al. 94] Eric Amiel, Olivier Gruber, and Eric Simon. Optimizing Multi-Method Dispatch Using Compressed Dispatch Tables. In *Proceedings OOPSLA '94*, pages 244–258, Portland, OR, October 1994.

[Bobrow et al. 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices*, 28(Special Issue), September 1988.

[Chambers & Ungar 90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 150–164, June 1990.

[Chambers 92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 33–56, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[Chambers 93a] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report UW-CSE-93-03-05, Department of Computer Science and Engineering. University of Washington, March 1993. Revised, March 1997.

[Chambers 93b] Craig Chambers. Predicate Classes. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Chambers et al. 96] Craig Chambers, Jeffrey Dean, and David Grove. Whole-Program Optimization of Object-Oriented Languages. Technical Report UW-CSE-96-06-02, Department of Computer Science and Engineering. University of Washington, June 1996.

[Chen et al. 94] Weimin Chen, Volker Turau, and Wolfgang Klas. Efficient Dynamic Look-up Strategy for Multi-Methods. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 408–431, Bologna, Italy, July 1994. Springer-Verlag.

[Dean et al. 95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[Dean et al. 96] Jeffrey Dean, Greg DeFouw, Dave Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *OOPSLA'96 Conference Proceedings*, San Jose, CA, October 1996.

[DeFouw et al. 98] Greg DeFouw, David Grove, and Craig Chambers. Fast Interprocedural Class Analysis. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, January 1998.

[Driesen et al. 95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message Dispatch on Pipelined Processors. In *Proceedings ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.

[Dujardin 96] Eric Dujardin. Efficient Dispatch of Multimehtods in Constant Time with Dispatch Trees. Technical Report 2892, INRIA, Rocquencourt, France, 1996.

[Dujardin et al. 98] Eric Dujardin, Eric Amiel, and Eric Simon. Fast algorithms for compressed multimethod dispatch table generation. *ACM Transactions on Programming Languages and Systems*, 20(1):116–165, January 1998.

[Dussud 89] Patrick H. Dussud. TICLOS: An Implementation of CLOS for the Explorer Family. In *Proceedings OOPSLA '89*, pages 215–220, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[Ernst et al. 98] Michael D. Ernst, Craig S. Kaplan, and Craig Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *Proceedings ECOOP '98*, Brussels, Belgium, July 1998. Springer-Verlag.

[Fernandez 95] Mary F. Fernandez. Simple and Effective Link-Time Optimization of Modula-3 Programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 103–115, June 1995.

[Goldberg & Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addision-Wesley, Reading, MA, 1983.

[Gosling et al. 96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[Grove et al. 95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers. Profile-Guided Receiver Class Prediction. In *OOPSLA'95 Conference Proceedings*, pages 108–123, Austin, TX, October 1995.

[Grove et al. 97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object Oriented

Languages. In *OOPSLA'97 Conference Proceedings*, Atlanta, GA, October 1997.

[Hamer et al. 90] J. Hamer, J.G. Hosking, and W.B. Mugridge. A Method for Integrating Classification Within an Object-Oriented Environment. Technical Report Auckland Computer Science Report No. 48, Department of Computer Science, University of Auckland, October 1990.

[Hölzle et al. 91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 21–38, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[Hu & Tucker 71] T. C. Hu and A. C. Tucker. "Optimal Computer Search Trees and Variable-Length Alphabetical Codes. *SIAM Journal on Applied Mathematics*, 21(4):514–532, 1971.

[Hudak et al. 92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell, Version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[Hyafil & Rivest 76] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, May 1976.

[Johnson 86] Ralph E. Johnson. Type-Checking Smalltalk. In *Proceedings OOPSLA '86*, pages 315–321, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[Johnson et al. 88] Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *Proceedings OOPSLA '88*, pages 18–26, November 1988. Published as ACM SIGPLAN Notices, volume 23, number 11.

[Kaehler & Krasner 83] Ted Kaehler and Glenn Krasner. LOOM — Large Object-Oriented Memory for Smalltalk-80 Systems. In G. Krasner, editor, *Smalltalk-80 — Bits of History, Words of Advice*, chapter 14, pages 251–270. Addison-Wesley, 1983.

[Kiczales & Rodriguez 90] Gregor Kiczales and Luis Rodriguez. Efficient Method Dispatch in PCL. In *1990 ACM Conference on Lisp and Functional Programming*, pages 99–105. ACM, ACM Press, June 1990.

[Knuth 68] Donald E. Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley, Reading, Mass., 1968.

[Meyer 92] Bertrand Meyer. *Eiffel: the language*. Prentice-Hall, 1992.

[Milner et al. 97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[Morel & Renvoise 79] Etienne Morel and Claude Renvoise. Global Optimization by Suppression of Partial Redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[Mugridge et al. 91] Warwick B. Mugridge, John Hamer, and John G. Hosking. Multi-Methods in a Statically-Typed Programming Language. In P. America, editor, *Proceedings ECOOP '91*, LNCS 512, pages 307–324, Geneva, Switzerland, July 15-19 1991. Springer-Verlag.

[Nakamura et al. 96] Hiroaki Nakamura, Tamiya Onodera, and Mikio Takeuchi. Message dispatch using binary decision trees. Technical Report RT0137, IBM Research, Tokyo Research Laboratory, Kanagawa, Japan, March 1, 1996.

[Nelson 91] Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Plevyak & Chien 94] John Plevyak and Andrew A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proceedings OOPSLA '94*, pages 324–340, Portland, OR, October 1994.

[Shalit 96] Andrew Shalit, editor. *The Dylan Reference Manual*. Addison-Wesley, Reading, MA, 1996.

[Steele Jr. 90] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, Bedford, MA, 1990. Second edition.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language (second edition)*. Addision-Wesley, Reading, MA, 1991.

[Taivalsaari 93] Antero Taivalsaari. Object-oriented programming with modes. *Journal of Object-Oriented Programming*, pages 25–32, June 1993.

[Vitek et al. 97] Jan Vitek, Nigel Horspool, and Andreas Krall. Efficient Type Inclusion Tests. In *Proceedings OOPSLA '97*, Atlanta, GA, October 1997.

[Zendra et al. 97] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch Without Virtual Function Tables: The SmallEiffel Compiler. In *OOPSLA'97 Conference Proceedings*, Atlanta, GA, October 1997.