Selective Specialization for Object-Oriented Languages

Jeffrey Dean, Craig Chambers, and David Grove

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
{jdean,chambers,grove}@cs.washington.edu

Abstract

Dynamic dispatching is a major source of run-time overhead in object-oriented languages, due both to the direct cost of method lookup and to the indirect effect of preventing other optimizations. To reduce this overhead, optimizing compilers for object-oriented languages analyze the classes of objects stored in program variables, with the goal of bounding the possible classes of message receivers enough so that the compiler can uniquely determine the target of a message send at compile time and replace the message send with a direct procedure call. *Specialization* is one important technique for improving the precision of this static class information: by compiling multiple versions of a method, each applicable to a subset of the possible argument classes of the method, more precise static information about the classes of the method's arguments is obtained. Previous specialization strategies have not been selective about where this technique is applied, and therefore tended to significantly increase compile time and code space usage, particularly for large applications. In this paper, we present a more general framework for specialization in object-oriented languages and describe a goal-directed specialization algorithm that makes selective decisions to apply specialization to those cases where it provides the highest benefit. Our results show that our algorithm improves the performance of a group of sizeable programs by 65% to 275% while increasing compiled code space requirements by only 4% to 10%. Moreover, when compared to the previous state-of-the-art specialization scheme, our algorithm improves performance by 11% to 67% while simultaneously reducing code space requirements by 65% to 73%.

1 Introduction

Dynamic dispatching is a serious run-time performance bottleneck for programs written in an object-oriented style. The expense of dynamically-dispatched calls (also known as virtual function calls or message sends) arises both from the direct cost of performing method lookup and from the indirect opportunity cost of the loss of other optimizations, such as inlining and interprocedural analysis. To avoid these costs, optimizing compilers for object-oriented languages strive to statically bind as many message sends to called methods as possible. Static binding requires information about the possible classes that could be stored in each program variable, so that the set of invocable methods can be determined: if only one method can be invoked and no method lookup error is possible, the send can be statically bound, i.e., replaced with a direct procedure call to the target method. Once statically bound, the call becomes amenable to further optimizations such as inline expansion.

Appeared in *SIGPLAN'95: Conference on Programming Language Design and Implementation (PLDI '95)*, June 1995.

One technique for improving the precision of class information, and indirectly to support more static binding, is to compile multiple specialized versions of a method, each applicable to only a subset of the possible argument classes of the method. By choosing the specializing subsets judiciously, many message sends can be statically bound that could not be in the unspecialized version. Some compilers for object-oriented languages implement a restricted form of specialization called *customization*, in which a specialized version of a method is compiled for each possible receiver class, and methods are never specialized for arguments other than the receiver [Chambers et al. 89]. Although this strategy yields substantial performance improvements, it can also lead to a substantial increase in code size, especially for large programs with deep inheritance hierarchies and a large number of methods.

In this paper, we present a goal-directed algorithm that combines dynamic profile data with static information about where in the class hierarchy methods are defined to identify profitable specializations. The algorithm also is suitable for use in object-oriented languages with multi-methods, unlike customization. Our results show that the algorithm simultaneously increases program performance and decreases the compiled code space requirements over several alternative compilation and specialization strategies, including customization.

The next section provides an example motivating why specialization is an important optimization technique and discusses why existing specialization techniques are inadequate. Section 3 describes our algorithm for selective specialization, Section 4 provides performance results for our algorithm, comparing it to a variety of alternative schemes, and Section 5 discusses related work.

2 A Motivating Example

To illustrate the issues involved in specialization, Figure 1 presents an example of a `Set` class hierarchy, with different subclasses for different set representations.* The abstract base class provides operations to perform generic set operations such as union, intersection, and overlaps testing, and relies on dynamically-dispatched `do` and `includes` messages, with versions of these operations implemented in each subclass.

Although writing the code in this fashion, where many operations are factored into the abstract `Set` class, offers software engineering advantages, it leads to inefficient code if compiled naively. Compiling a single version of the `overlaps` method that is general enough to apply to all possible `Set` representations results in code that has substantial dynamic dispatching overhead. In such code, the

* In our syntax, `λ(arg1:type){` ... *code* ... `}` is a closure that accepts one argument, and `λ(type):type` is a static type declaration for such a closure. `self` is the name of the receiver of a method. Other arguments of the form `Class::arg` represent methods that are dispatched on multiple arguments (multi-methods). Alternatively, in a singly-dispatched language, they could be written in a double-dispatching style [Ingalls 86]. Dynamically-dispatched message sends are shown in ***`this`*** font.

```
class Set[T] {
  method overlaps(set2:Set[T]):bool {
      self.do(λ(elem:T){ if set2.includes(elem) then return true; });
      return false;
  }
  method includes(elem:T):bool {
      -- A default includes implementation: subclasses can override to provide a more efficient implementation
      self.do(λ(e2:T){ if elem.equal(e2) then return true; });
      return false;
  }
  .. other set operations such as intersection, union, etc. ...
}
class ListSet[T] subclasses Set[T] {
  method do(body:λ(T):void):void {
      ... code to iterate over elements of t, evaluating closure body on each element ...
  }
}
class HashSet[T] subclasses Set[T] {
  method do(body:λ(T):void):void { ... }
  method includes(elem:T):bool {
      -- A more efficient implementation of includes: hash the element and see if it is in that bucket
  }
}
class BitSet[T] subclasses Set[T] {
  method do(body:λ(T):void):void { ... }
  method includes(elem:T):bool {
      -- A more efficient implementation of includes: test the appropriate bit position
  }
  ... Provide more efficient versions of overlaps, union, intersection, etc. when operating on two BitSets
  method overlaps(BitSet[T]::set2):bool { ... }
}
```

Figure 1: A set abstraction hierarchy

`self.`***`do`***`(λ(elem){...})` message must be dynamically dispatched, since which implementation of `do` will be invoked cannot be determined statically and in fact can vary at run-time. Furthermore, within the closure, the send of `set2.`***`includes`***`(elem)` must also be dynamically dispatched, since there is insufficient static information about the class of `set2` to enable static binding to a single `includes` method. The direct cost of performing the dynamic dispatching is substantial, but having the messages be dynamically dispatched also prevents other optimizations, such as inlining, which often has an even greater effect on code quality. For example, because the ***`do`*** message send was not able to be statically-bound and inlined, the closure argument to ***`do`*** must be created at run-time and invoked as a separate procedure for each iteration.

Customization is a simple specialization scheme used in the implementations of some object-oriented languages, including Self [Chambers & Ungar 89, Hölzle & Ungar 94], Sather [Lim & Stolcke 91], and Trellis [Kilian 88]: a specialized version of each method is compiled for each class inheriting the method. Within the customized version of a method, the exact class of the receiver is known, enabling the compiler to statically bind messages sent to the receiver formal parameter (`self`). In our example, a specialized version of `overlaps` would be compiled for `ListSet` and `HashSet`, and this would allow the `self.`***`do`***`(λ(elem){...})` to be statically-bound to the appropriate implementation in each of these versions (`BitSet` provides an overriding version of `overlaps`, and therefore uses this method definition for customization purposes). For example, in the `ListSet` version of `overlaps`, the `self.`***`do`***`(λ(elem){...})` message can get inlined down to a simple `while` loop. Other operations defined on `Set`, such as `union` and `intersection`, would be similarly customized. Because sends to `self` tend to be fairly common in object-oriented programs, customization is effective at increasing execution performance: Self code runs 1.5 to 5 times faster as a result of customization, and customization was one of the single most important optimizations included in the Self compiler [Chambers 92]. Lea hand-simulated customization in C++ for a Matrix class hierarchy, showing an order-of-magnitude speedup, and argued for the inclusion of customization in C++ implementations [Lea 90].

Unfortunately, this simple strategy for specialization suffers from the twin problems of *overspecialization* and *underspecialization*, because specialization is done without regard for its costs and benefits. In many cases, multiple specialized versions of a method are virtually identical and could be coalesced without a significant impact on program performance. For large programs with deep inheritance hierarchies and many methods, producing a specialized version of every method for every potential receiver class leads to serious code explosion. In the presence of large, reusable libraries, we expect applications to use only a subset of the available classes and operations, and some of those only infrequently, and consequently simple customization is likely to be impractical.

In systems employing dynamic compilation, such as the Self system [Chambers & Ungar 89], customization can be done lazily by delaying the creation of a specialized version of a method until the particular specialized instance is actually needed at runtime, if at all. This strategy avoids generating code for class × method combinations that are never used, but such systems can still have problems with overspecialization if a method is invoked with a large number of distinct receiver classes during a program's execution or if a method is invoked only rarely for particular receiver classes. Moreover, dynamic compilation may not be a suitable framework for some programming systems because of the need for an optimizing compiler and some representation of the program source to be included in the program's runtime system.

In addition to overspecialization, the simple customization approach suffers from *underspecialization*, because methods are never specialized on arguments other than the receiver. In many cases, considerable benefits can arise from specializing a method for particular argument classes. In our example, specializing the `overlaps` method for a particular class of the `set2` argument could have considerable performance benefits, since it would allow the `set2.`***`includes`***`(elem)` message inside the loop to be statically-bound and potentially inlined. By specializing on an argument other than the receiver, clients of `overlaps` will have to be compiled to choose the appropriate specialized version of `overlaps`, perhaps by inserting additional class tests or table lookups at run-time, but such tests occur only once per `overlaps` operation. Compared to incurring a dynamic dispatch for every element of the set, this overhead to select the appropriate specialization can be minor. In effect, specialization can hoist dynamic dispatches from frequently executed sections of code to less frequently executed sections, often across multiple procedure boundaries.

3 Selective Specialization

To address the problems of overspecialization and underspecialization, our work focuses on techniques for specializing only where the benefits are large. Rather than specializing exhaustively, our algorithm exploits dynamic profile data to specialize heavily-used methods for their most beneficial argument classes, preserving a single, general-purpose version of the method that works for the remaining cases. Moreover, our algorithm identifies when a specialization can be shared by a group of classes without great loss in performance, further reducing code space costs.

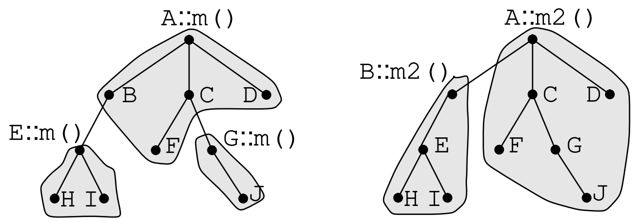
Our algorithm is based on a general framework in which a method can be specialized for a tuple of class sets, one class set per formal argument, including the receiver. For example, in the `Set` example we would describe one potential specialization of the `overlaps` method using the tuple `<{ListSet,HashSet},{HashSet}>`. An argument class set can include multiple classes when a specialization is shared by several classes. This framework has two advantages over previous schemes:

- Methods can be specialized to obtain more precise information about any formal argument, not just the receiver argument as with customization.
- Multiple classes can share a single specialized method, rather than generating a specialized copy for each individual class.

However, this general model places few constraints on specialization, requiring guidance in order to determine which potential specializations are profitable. Our algorithm relies on two sources of information to guide the specialization process:

- We exploit information about the classes and methods defined in the program to identify groups of classes that still allow static binding of message.
- We use `gprof`-style profile information to select the most important specializations in program hot spots.

The algorithm will be illustrated with the class hierarchy and method definitions shown in Figure 2 and the weighted call graph




```
method A::m3(arg2:A) { self.m4(arg2); }
method A::m4(arg2:A) { self.m(); arg2.m2(); }
```

Figure 2: Example class hierarchy

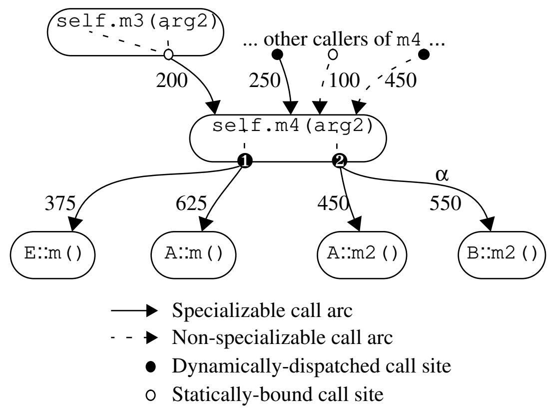


Figure 3: Example call graph

shown in Figure 3. The class hierarchy consists of nine classes, with method `m()` implemented only in classes `A`, `E`, and `G`, method `m2()` implemented only in classes `A` and `B`, and methods `m3` and `m4` implemented only in class `A`. The shaded regions show the "equivalence classes" that all invoke the same implementation of `m` and `m2` (note that the illustrations in Figure 2 are two views of the same underlying inheritance hierarchy).

The algorithm is shown in Figure 1. Set operations on tuples are defined to operate pairwise on the tuple elements. The algorithm relies on the presence of a weighted program call graph constructed from the profile data that describes, for each call site in the program, the set of methods invoked and the number of times each was invoked (a call site can have multiple arcs due to dynamic dispatching). Given an arc in the call graph, *Caller*(*arc*) gives the calling method, *Callee*(*arc*) gives the called method, *CallSite*(*arc*) identifies the message send site within the caller, and *Weight*(*arc*) gives the execution count of the arc. For the arc labelled α in Figure 3, *Caller*(α) is `A::m4`, *Callee*(α) is `B::m2`, *CallSite*(α) is the send `arg2.`***`m2`***`()` within `m4`, and *Weight*(α) is 550.

The algorithm exploits information about the class hierarchy through the *ApplicableClasses* function. *ApplicableClasses*⟦*meth*⟧ returns the tuple of the set of classes for each formal argument for which the method *meth* could be invoked (excluding classes that bind to overriding methods). The shaded "equivalence classes" regions in the example above identify the *ApplicableClasses* for each of the `m` and `m2` methods. For example, *ApplicableClasses*⟦`method E::m()`⟧ = `<{E,H,I}>`. For singly-dispatched languages, computing *ApplicableClasses* for each method is fairly straightforward. For multiply-dispatched languages, there are some subtleties in performing this computation efficiently; details of how this can be done can be found elsewhere [Dean et al. 95].

Profile info:

Caller(*a*), *Callee*(*a*), *CallSite*(*a*), and *Weight*(*a*) give caller, callee, call site, and execution count for arc *a*. in call graph.

Source info*:*

ApplicableClasses⟦`method m(f`$_1$`, f`$_2$`, ..., f`$_n$`)` ⟧= *n*-tuple of sets of classes for `f`$_1$`, f`$_2$`, ..., f`$_n$ for which `m` might be invoked

PassThroughArgs⟦`msg(arg`$_1$`, arg`$_2$`, ..., arg`$_n$`)` *in* `method m(f`$_1$`, f`$_2$`, ..., f`$_m$`)` ⟧= {<*fpos*→ *apos* > | `f`$_{fpos}$ = `arg`$_{apos}$}

e.g. PassThruArgs⟦`f2.foo(x, f1)` *in* `method m(f1, f2)` ⟧= {<1→3>,<2→1>} (the receiver is argument 1)

Input: *SpecializationThreshold*, the minimum *Weight*(*arc*) for an arc to be considered for specialization

Output: $Specializations_{meth}$: set of tuples of sets of classes for which method *m* should be specialized

specializeProgram() =
  foreach method *meth* **do**
    $Specializations_{meth}$:= *ApplicableClasses*⟦*meth*⟧;
  foreach method *meth* **do**
    specializeMethod(*meth*);

specializeMethod(*meth*) =
  foreach *arc* s.t. *Caller*(*arc*) = *meth* **and** *isSpecializableArc*(*arc*) **do**
    if *Weight*(*arc*) > *SpecializationThreshold* **then**
      addSpecialization(*meth*, *neededInfoForArc*(*arc*));

isSpecializableArc(*arc*) **returns** bool **=**
  return *PassThroughArgs*⟦*CallSite*(*arc*)⟧≠ ∅ **and** *ApplicableClasses*⟦*Caller*(*arc*)⟧≠ *neededInfoForArc*(*arc*);

neededInfoForArc(*arc*) **returns** Tuple[Set[Class]] =
  return *neededInfoForArc*(*arc*, *ApplicableClasses*⟦*Callee*(*arc*)⟧);

neededInfoForArc(*arc, calleeInfo*) **returns** Tuple[Set[Class]] =
  needed := *ApplicableClasses*⟦*Caller*(*arc*)⟧;
  foreach <*fpos*→ *apos*> ∈ *PassThroughArgs*⟦*CallSite*(*arc*)⟧**do**
    $needed_{fpos}$:= $needed_{fpos}$ ∩ $calleeInfo_{apos}$;
  return *needed*;

addSpecialization(*meth*, *specTuple*) =
  foreach *existingSpec* ∈ $Specializations_{meth}$ **do**
    if *specTuple* ∩ *existingSpec* ≠ ∅ **then**
      $Specializations_{meth}$:= $Specializations_{meth}$ ∪ (*existingSpec* ∩ *specTuple*);
  foreach *arc* s.t. *Callee*(*arc*) = *meth* **do**
    cascadeSpecializations(*arc*, *spec*);

cascadeSpecializations(*arc*, *calleeSpec*) =
  if *PassThroughArgs*⟦*CallSite*(*arc*)⟧≠ ∅ **and** *ApplicableClasses*⟦*Caller*(*arc*)⟧= *neededInfoForArc*(*arc*) **and**
      Weight(*arc*) > *SpecializationThreshold* **then**
    callerSpec := *neededInfoForArc*(*arc*, *calleeSpec*);
    if *callerSpec* ≠ ∅ **and** *callerSpec* ∉ $Specializations_{Caller(arc)}$ **then**
      addSpecialization(*Caller*(*arc*), *callerSpec*);

Figure 4: Selective specialization algorithm

The goal of the algorithm is to eliminate dynamic dispatches for calls from some method by specializing that method for particular classes of its formal parameters. By providing more precise information about the classes of a method's formals, the algorithm attempts to make more static information available to dynamically-dispatched call sites within the method to enable the call sites to be statically bound in the specialized version. The simplest case in which specializing a method's formal can provide better information about a call site's actual occurs when the formal is passed directly as an actual parameter in the call; we call such a call site a *pass-through* call site (a similar notion is found in the jump functions of Grove and Torczon [Grove & Torczon 93]). Our algorithm focuses on pass-through dynamically-dispatched call sites, which we term *specializable call sites*, as the sites that can potentially benefit from specialization. For the `overlaps` example, the sends of ***`do`*** and ***`includes`*** are specializable call sites, since obtaining more precise information about the classes of `self` and `set2` can enable static binding of these message sends.

In summary, our algorithm visits each method in the call graph. The algorithm searches for high-weight, dynamically-dispatched, pass-through calls from the method, i.e., those call arcs that are both possible and profitable to optimize through specialization of the enclosing method. The algorithm computes the greatest subset of classes of the method's formals that would support static binding of the call arc, and creates a corresponding specialization of the method if such a subset of classes exists.

Much of the remainder of this section examines key aspects of the algorithm in more detail, focusing on the following issues:

- How is the set of classes that enable specialization of a call arc computed? This is computed by the *neededInfoForArc* function, as discussed in Section 3.1.
- How should specializations for multiple call sites in the same method be combined? This is handled by the *addSpecialization* routine and is discussed in Section 3.2.
- If a method *m* is specialized, how can we avoid converting statically-bound calls to *m* into dynamically-bound calls? Cascading specializations upwards (the *cascadeSpecializations* routine) can solve the problem in many cases, and is discussed in Section 3.3.
- When is an arc important to specialize? The algorithm currently uses a very simple heuristic, and Section 3.4 discusses the tradeoffs involved.
- At run-time, how is the right specialized version chosen? This is discussed in Section 3.5.
- What is the interaction between the specialization algorithm and first-class nested functions? Section 3.6 considers this question.

This section concludes with a discussion of some related issues:

- How does the need to examine the whole class hierarchy interact with separate compilation?
- How expensive is gathering and managing the necessary profile information?
- Can specialization be adapted to work in a system based on dynamic compilation?

3.1 Computing Specializations for Call Sites

The algorithm visits each high-weight pass-through arc leaving a method. For each such arc, it determines the most general class set tuple for the pass-through formals that would allow static binding of the call arc to the callee method. This information is computed by the *neededInfoForArc* function, which maps the *ApplicableClasses* for the callee routine back to the caller's formals using the mapping contained in the *PassThroughArgs* for the call site; if no specialization is possible, then the caller's *ApplicableClasses* is returned unchanged. As an example, consider arc α from the call graph in Figure 3. For this arc, the caller's *ApplicableClasses* is <{A,B,..., J},{A,B,..., J}>, the callee's *ApplicableClasses* tuple is <{B,E,H,I}>, and the *PassThroughArgs* mapping for the call site is {<2→1>}, so *neededInfoForArc*(α) is <{A,B,..., J}, {B,E,H,I}>. This means that within the specialized version, the possible classes of `arg2` are restricted to be in {B,E,H,I}; this information is sufficient to statically-bind the message send of `m2` to `B::m2()` within the specialized version. The *neededInfoForArc*(α) tuple will be added to the set of specializations for `m4` ($Specializations_{\mathtt{m4}}$).

3.2 Combining Specializations for Distinct Call Sites

Different call arcs within a single method may generate different class set tuples for specialization. These different tuples need to be combined somehow into one or more specialization tuples for the method as a whole. Deciding how to combine specializations for different call sites in the same method is a difficult problem. Ideally, the combined method specialization(s) would cover the combinations of method arguments that are most common and that lead to the best specialization, but it is impossible, in general, to examine only the arc counts in the call graph and determine what argument tuples the enclosing method was called with. In our example, we can determine that within `m4`, the class of `self` was in {A,B,C,D,F} 625 times and in {E,H,I} 375 times, and that the class of `arg2` was in {B,E,H,I} 550 times and in {A,C,D,F,G,J} 450 times, but we cannot tell in what combinations the argument classes appeared.

Because we want to be sure to produce specializations that help the high-benefit call arcs, our algorithm "covers all the bases," producing method specializations for all plausible combinations of arc specializations. This is the function of the *addSpecialization* function: given a new arc specialization tuple, it forms the combination of this new tuple with all previously-computed specialization tuples (including the initial unspecialized tuple) and adds these new tuples to the set of specializations for the method. For example, given two method specialization class set tuples $<A_1, \ldots, A_n>$ and $<A_1 \cap B_1, \ldots, A_n \cap B_n>$, adding a new arc specialization tuple $<C_1, \ldots, C_n>$ leads to four method specialization tuples for the method: $<A_1, \ldots, A_n>$, $<A_1 \cap B_1, \ldots, A_n \cap B_n>$, $<A_1 \cap C_1, \ldots, A_n \cap C_n>$, and $<A_1 \cap B_1 \cap C_1, \ldots, A_n \cap B_n \cap C_n>$ (assuming none of these intersections are empty: tuples containing empty class sets are dropped). For the example in Figures 2 and 3, nine versions of `m4` would be produced, including the original unspecialized version, assuming that all four outgoing call arcs were above threshold.

Although this approach can in principle produce a number of specializations for a particular method that is exponential in the number of specializable call arcs emanating from the method, we have not observed this behavior in practice. For the benchmarks described in Section 4, we have observed an average of 1.9 specializations per method receiving any specializations, with a maximum of 8 specializations for one method. We suspect that in practice we do not observe exponential blow-up because most call sites have only one or two high-weight specializable call arcs and because methods tend to have a small number of formal arguments that are highly polymorphic. Were exponential blow-up to become a problem, the profile information could be extended to maintain a set of tuples of classes of the actual parameters passed to each method during the profiling run. Given a set of potential specializations, the set of actual tuples encountered during the profiling run could be used to see which of the specializations would actually be invoked with high frequency. Of course, it is likely to be more expensive to gather profiles of argument tuples than to gather simple call arc and count information.

3.3 Cascading Specializations

Before a method is specialized, some of its callers might have been able to statically-bind to the method. When specialized versions of a method are introduced, however, it is possible that the static information at the call site will not be sufficient to select the appropriate specialization. This is the case with the arc from `m3` to `m4` in the example: `m3` had static information that both its arguments were descendents of class A, which was sufficient to statically-bind to `m4` when there was only a single version of `m4`, but not after multiple versions of `m4` are produced.

In such a case, there are two choices: these statically-bound calls could be left unchanged, having them call the general-purpose version of the routine, or the statically-bound call could be replaced with a dynamically-bound call that selects the appropriate specialization at run-time. The right choice depends on the amount of optimization garnered through specialization of the callee relative to the increased cost of dynamically dispatching the call to the specialized method.

In some cases, this conversion of statically-bound calls into dynamically-dispatched sends can be avoided by specializing the calling routine to match the specialized callee method. This is the purpose of the *cascadeSpecializations* function. Given a specialization of a method, it attempts to specialize statically-bound pass-through callers of the method to provide the caller with sufficient information to statically-bind to the specialized version of the method. *cascadeSpecializations* first checks to make sure the call arc was statically bound (with respect to the pass-through arguments) and of high weight; if the call arc is dynamically bound, then regular specialization through *specializedMethod* will attempt to optimize that arc. For example, when specializing the `m4` method for the tuple `<{A,B,C,D,F}, {A,C,D,F,G,J}>`, the arc from the `m3` method is identified as a target for cascaded specialization, but none of the other three callers are. If the calling arc passes this first test, *cascadeSpecializations* computes the class set tuple for which the caller should be specialized in order to support static binding of the call arc to the specialized version. For this example, the computed class set tuple for `m3` (*callerSpec* in the algorithm) is `<{A,C,D,F}, {A,C,D,F}>`. If the algorithm determines that the call site can call the specialized version, and that specialization of the caller is necessary to enable static binding, the algorithm recursively specializes the caller method (if that specialization hasn't already been created). This recursive specialization can set off ripples of specialization running upwards through the call graph along statically-bound pass-through high-weight arcs. Recursive cycles of statically-bound pass-through arcs do not need special treatment, other than the check to see whether the desired specialization has already been created.

3.4 Improved Cost-Benefit Analysis

The algorithm as presented currently uses a very simple heuristic for deciding what to specialize: if the weight of a specializable arc is larger than the *specializationThreshold* parameter, then the arc will be considered for specialization. In our implementation, the *specializationThreshold* is 1,000 invocations. There are several shortcomings of this simple approach. First, the code space increase incurred by specializing is not considered. A more intelligent heuristic could compute the set of specializations that would be necessary to statically bind a particular arc, and factor this information into the decision-making process. Second, it treats all dynamic dispatches as equally costly. Due to the indirect effects of optimizations such as inlining, the benefits of statically binding some message sends can be much higher than others. A more sophisticated heuristic could estimate the performance benefit of static binding, taking into account post-inlining optimizations [Dean & Chambers 94]. Third, the heuristic has no global view on the consumption of space during specialization. Alternatively, the algorithm could be provided with a fixed space budget, and could visit arcs in decreasing order of weight, specializing until the space budget was consumed.

We initially implemented the simple heuristic using the *specializationThreshold* to get the algorithm up and running, but had expected to implement a more sophisticated heuristic later that would make better time/space tradeoff decisions. However, in our system, the tradeoff implemented by the simple heuristic has been more than adequate, as indicated by the empirical results reported in Section 4.

3.5 Selecting the Appropriate Version of a Specialized Method

At run-time, message lookup needs to select the appropriate specialized version of a method. In singly-dispatched languages like C++ and Smalltalk, existing message dispatch mechanisms work fine for selecting among methods that are specialized only on the receiver argument. Allowing specialization on arguments other than the receiver, however, can create *multi-methods* (methods requiring dispatching on multiple argument positions) in the implementation, even if the source language allows only singly-dispatched methods. If the runtime system does not already support multi-methods, it must be extended to support them. A number of efficient strategies for multi-method lookup have been devised, including trees of single dispatch tables [Kiczales & Rodriguez 89], compressed multi-method dispatch tables [Chen et al. 94, Amiel et al. 94], and polymorphic inline caches extended to support multiple arguments [Hölzle et al. 91]. The choice of a multi-method dispatching mechanism is orthogonal to our specialization algorithm, however.

3.6 Specializing Nested Methods

In the presence of lexically-nested first-class functions, a message can be sent to a formal not of the outermost enclosing method but to a lexically-nested function (such as a closure that accepts arguments). Although our current implementation only considers the outermost method for specialization, in principle there is no reason that the algorithm could not produce multiple specialized versions of nested methods. At run-time, the representation of a closure could not contain a direct pointer to the target method's code, but instead would contain the address of a stub routine that performed the necessary dispatching for the specialized argument positions.

3.7 Other Issues

In this subsection we address a variety of issues relating to the implementation of the algorithm.

3.7.1 Whole Program Analysis and Incremental Compilation

Examination of the complete class hierarchy of the program, an underlying component of our specialization algorithm, might seem to be in conflict with incremental compilation: the compiler generates code containing embedded assumptions about the structure of the program's class inheritance hierarchy and method definitions, and these assumptions might change whenever the class hierarchy is altered or a method is added or removed. (The algorithm does not depend on the implementations of the methods, other than the pass-through information about call sites.) To help provide incremental compilation in the face of this whole-program analysis, our compiler maintains fine-grained dependency information to selectively recompile those pieces of the program that are invalidated as a result of some change to the class hierarchy or the set of methods in the program. The dependency information forms a directed, acyclic graph, with nodes representing pieces of information, and edges representing dependencies. The dependency graph is constructed incrementally during compilation. Whenever a portion of the compilation process uses a piece of information that could change, the compiler adds an edge to the dependency graph from the node representing the information used to the node representing the client of the information. When changes are made to the source program, the compiler computes what source dependency nodes have been affected and propagates invalidations downstream from these nodes. This invalidates all information (including compiled code modules) that depended on the changed source information. Further details of this dependency representation and measurements of the amount of recompilation required for a 3-week period of programming can be found elsewhere [Chambers et al. 95].

3.7.2 Gathering and Managing Profile Information

Obtaining the profile data needed by the specialization algorithm requires that the program executable be built with the appropriate instrumentation. To enable long profiling runs and profiling of typical application usage, profiling should be as inexpensive as possible, since otherwise it may not be feasible to gather profile information. Computing counts for statically-bound arcs can be done by inserting counters at statically-bound call sites. The expense of profiling dynamically-dispatched message sends depends in large part on the run-time system's message dispatching mechanism. Some systems, including the Cecil system in which we implemented the algorithm, use *polymorphic inline caches* [Hölzle et al. 91]: call-site-specific association lists mapping receiver classes to target methods. To gather call-site-specific profile data, counter increments are added to each of the cases, and the counters for the PICs of all call sites are dumped to a file at the end of a program run. The run-time overhead of this profiling for our Cecil benchmarks (described in Section 4) is 15-50%. In other systems that use method dispatching tables, such as C++ and Modula-3 implementations, additional code could be inserted to maintain profile information at message send sites.

To make managing profile information more convenient, our compiler maintains a persistent internal database of profile information that is consulted transparently during compilations. Additionally, for object-oriented programs we have observed that the kind of profile information needed to construct this call graph remains fairly constant across different inputs to a program and even as the program evolves [Garrett et al. 94], so profiling can be done relatively infrequently and reused across many compilations.

3.7.3 Applicability to a Dynamic Compilation Environment

Our algorithm is suitable for use in a dynamic compilation environment such as Self. The Self system initially compiles methods without optimization but installs counters to detect the heavily-used methods. When a counter exceeds a threshold, the system recompiles portions of the code using optimization to adapt the code to program hot spots. Our specialization algorithm could be used in such a system. The unoptimized code would keep track of the target methods for each call site and the counts (essentially arcs in the call graph), and the appropriate localized portion of the call graph could be constructed as necessary to make specialization decisions during the recompilation process. Section 4 provides results showing that, even for a system that compiles code lazily, our approach could lead to a significant reduction in code space requirements over a system that employs simple customization.

4 Performance Evaluation

To evaluate the effectiveness of our algorithm, we implemented several different specialization schemes in the context of the Vortex compiler for Cecil, a pure object-oriented language based on multi-methods [Chambers 93]. The implementation of our algorithm constructs a weighted call graph from profiles of the program and then generates a list of specialization directives using our algorithm. The compiler then executes the directives to produce the specialized versions of methods.

We compared our specialization scheme with several alternative schemes, described in Table 1. Our base configuration performs

Table 1: Compiler Configurations

Configuration	Description
Base	Intraprocedural class analysis, inlining, splitting, CSE & constant propagation & folding, dead code elimination (to optimize away unneeded closure creations), and hard-wired class prediction for a small number of common messages such as `if` and `+`. This set of optimizations is roughly comparable to that performed by the Self-91 compiler [Chambers & Ungar 91]. The compiler produces one compiled version for each source method.
Cust	**Base** + simple customization: specialize each method for each inheriting class for the receiver argument. This corresponds to the approach take by the implementations of Sather, Trellis, and Self.
Cust-MM	**Base** + customization extended to multi-methods by customizing on each possible combination of dispatched arguments. The code space requirements of this configuration make it impractical for statically-compiled systems
CHA	**Base** + class hierarchy analysis. The class hierarchy of the whole program is examined to enable conversion of dynamically-bound calls to statically-bound calls when the compiler detects that there are no overriding methods [Dean et al. 95].
Selective	**CHA** + our profile-guided selective specialization algorithm.

significant intraprocedural optimizations, seeking to temper the effects of Cecil's pure language model. We measured the effectiveness of these schemes in terms of both code space used and runtime performance on four Cecil benchmark programs, described in Table 2.

Table 2: Benchmarks

Program	Lines[a]	Description
Richards	400	Operating system task queue simulation
InstSched	2,400	A MIPS assembly code instruction scheduler
Typechecker	17,000[b]	Typechecker for the Cecil language
Compiler	37,500[b]	Optimizing compiler for the Cecil language

a. In addition to the 8,500-line standard library
b. The typechecker and compiler share approximately 12,000 lines of code.

To evaluate the performance of the algorithm, we measured both the dynamic number of dynamic dispatches in each version of the programs as well as the bottom-line execution speed. The results are shown in Figure 5. All data have been normalized to the **Base** configuration for each program. For the bottom-line execution speed graph, taller bars indicate better performance; for the code space usage graphs, shorter bars indicate more compact compiled code.

For the selective algorithm, we used one set of inputs to the Typechecker and Compiler benchmarks for gathering the profiles and a different set of inputs for measuring the resulting specialized

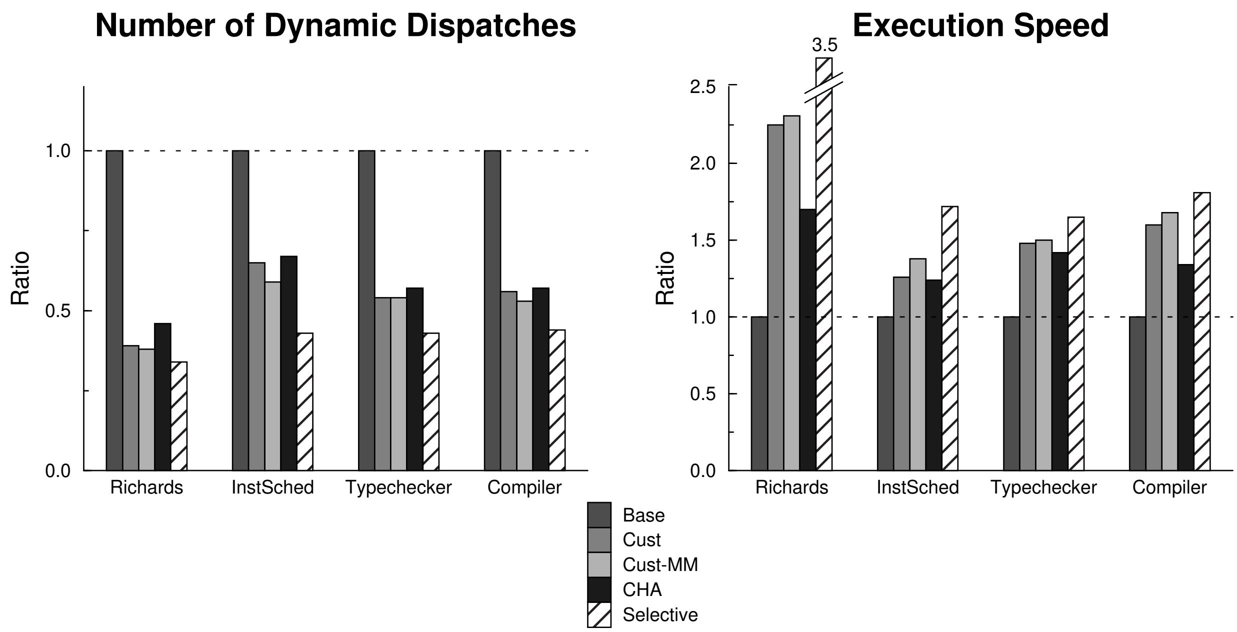


Figure 5: Number of dynamic dispatches and execution speed

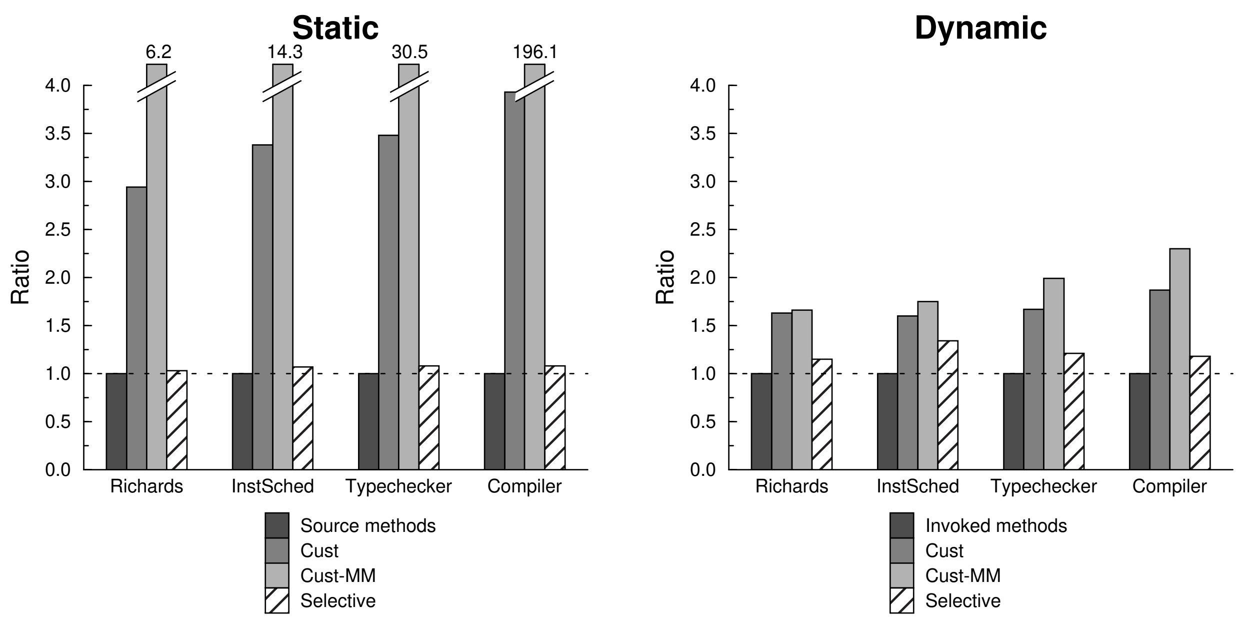


Figure 6: Number of routines compiled (static and dynamic)

programs. In practice, the choice of input did not seem to have a significant impact on performance, however.

Simple customization eliminated 35-61% of the dynamic dispatches in the benchmark programs over the baseline. Customizing for all combinations of dispatched arguments for multi-methods eliminated slightly more dispatches, ranging from 41-62%. Selective specialization did best of all, eliminating 54-66% of the dispatches; class hierarchy analysis, a necessary component of our selective specialization algorithm, alone eliminated 33-54% of the dispatches.

Translated to bottom-line performance measurements, customization sped up the programs by 26-125% (by 38-131% if customizing on all dispatched arguments). Our selective specialization algorithm performed considerably better, speeding the programs by 65-275%. Class hierarchy analysis by itself sped up the programs by 24-70%, accounting for roughly a third of the benefit attributable to our algorithm.

The space requirements for the various versions of the benchmarks are summarized in Figure 6.* We present two different versions of the space requirements: the first is the number of specialized methods that are produced in a statically-compiled system such as Sather,

* The runtime performance values for **Cust-MM** were derived from an executable that contained only those specialized methods that were invoked during program execution. **Cust-MM** is practical only for dynamic compilation systems, due to its prohibitive code space increase in statically-compiled systems.

Trellis, or Cecil, and the second is the number that would be produced in a dynamic compilation system such as Self, where only the methods invoked at run-time get compiled and specialized. Compile time numbers are not reported, but the compile time required should be roughly proportional to the number of specialized methods compiled.

Our selective specialization algorithm produces fewer specializations than either of the simple customization strategies. In statically-compiled systems, receiver-oriented customization increases the code space required for the benchmarks by a factor of three to four. Our selective specialization algorithm increased code space by only 4-10% for the benchmarks, despite achieving the highest run-time performance. For larger programs, selective specialization may be the only practical specialization strategy in a statically-compiled system.

5 Related Work

The implementations of Self [Chambers & Ungar 91], Trellis [Kilian 88], and Sather [Lim & Stolcke 91] use customization to provide the compiler with additional information about the class of the receiver argument to a method, allowing many message sends within each customized version of the method to be statically-bound. All of this previous work takes the approach of always specializing on the exact class of the receiver and not specializing on any other arguments. The Trellis compiler merges specializations after compilation to save code space, if two specializations produced identical optimized code; compile time is not reduced, however. As discussed in Section 2, customization can lead to both overspecialization and underspecialization. Our approach is more effective because it identifies sets of receiver classes that enable static binding of messages and uses profile data to ignore infrequently-executed methods (thereby avoiding overspecialization), and because it allows specialization on arguments other than just the receiver of a message (preventing underspecialization for arguments).

Cooper, Hall, and Kennedy present a general framework for identifying when creating multiple, specialized copies of a procedure can provide additional information for solving interprocedural dataflow optimization problems [Cooper et al. 92]. Their approach begins with the program's call graph, makes a forward pass over the call graph propagating "cloning vectors" which represent the information available at call sites that is deemed interesting by the called routine, and then makes a second pass merging cloning vectors that lead to the same optimizations. The resulting equivalence classes of cloning vectors indicate the specializations that should be created. Their framework applies to any forward dataflow analysis problem, such as constant propagation. Our work differs from their approach in several important respects. First, we do not assume the existence of a complete call graph prior to analysis. Instead, we use a subset of the real call graph derived from dynamic profile information. This is important, because a precise call graph is difficult to compute in the presence of extensive dynamically dispatched messages [Palsberg & Schwartzbach 91, Plevyak & Chien 94, Pande & Ryder 94]. Second, our algorithm is tailored to object-oriented languages, where the information of interest is derived from the specializations of the arguments of the called routines. Our algorithm consequently works backwards from dynamically-dispatched pass-through call sites, the places that demand the most precise information, rather than proceeding in two phases as does Cooper *et al.*'s algorithm. Finally, our algorithm exploits profile information to select only profitable specializations.

Procedure specialization has been long incorporated as a principal technique in partial evaluation systems [Jones et al. 93]. Ruf, Katz, and Weise [Ruf & Weise 91, Katz & Weise 92] address the problem of avoiding overspecialization in their FUSE partial evaluator. Their work seeks to identify when two specializations generate the same code. Ruf identifies the subset of information about arguments used during specialization and reuses the specialization for other call sites that share the same abstract static information. Katz extends this work by noting when not all of the information conveyed by a routine's result is used by the rest of the program. Our work differs from these in that we are working with a richer data and language model than a functional subset of Scheme and our algorithm exploits dynamic profile information to avoid specializations whose benefits would be accrued only infrequently.

6 Conclusions

We are exploring a number of areas in this line of research. First, we are working on applying the specialization algorithm to other languages, such as C++ and Modula-3, to examine its performance across a variety of language models. We are also investigating ways of specializing callers for the return values of the called methods, so that knowledge of the class of the return value can be propagated to the caller, perhaps by returning to different locations based on the class of the return value.

On a larger scale, we are exploring how different optimization techniques for optimizing object-oriented programs interact. Specialization is only one technique for reducing dynamic-dispatching overhead in object-oriented programs. A number of other techniques have also demonstrated substantial performance improvements for programs with extensive dynamic dispatching, including whole-program class hierarchy analysis [Dean et al. 95], profile-guided class prediction [Garrett et al. 94, Hölzle & Ungar 94], and interprocedural class inference [Palsberg & Schwartzbach 91, Plevyak & Chien 94]. All of these techniques are striving to eliminate dynamic dispatches and indirectly enable other optimizations, such as inlining, so it seems clear that the performance benefits of combining all of these techniques will not be strictly additive. We are currently exploring how these different optimization techniques interact.

We have presented a general framework for specialization in object-oriented languages and an algorithm that combines static analysis and profile information to identify the most profitable specializations. Compared to a base configuration that does no specialization, our algorithm improves the performance of a suite of realistic programs ranging from 400 to 50,000 lines by 65% to 275% while increasing compiled code space requirements by only 4% to 10%. Compared to customization, the previous state-of-the-art specialization technique for object-oriented languages included in several language implementations, our algorithm *improves* performance by 20% to 68%, while simultaneously *reducing* code space requirements by 65% to 73%. As a consequence of its more judicious application of specialization, our algorithm is appropriate for specializing on multiple arguments of a method and for use in object-oriented languages with multi-methods.

Acknowledgments

This research is supported in part by a NSF Research Initiation Award (contract number CCR-9210990), a NSF Young Investigator Award (contract number CCR-9457767), a grant from the Office of Naval Research (contract number N00014-94-1-1136), and gifts from Sun Microsystems, IBM, Pure Software, and Edison Design Group. Mark Linton and the anonymous referees provided helpful comments that improved the presentation of this paper. Stephen North and Eleftherios Koutsofios of AT&T Bell Laboratories provided us with `dot`, an automatic graph layout program. An early version of this work appeared in the 1994 Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94).

Other papers on the Cecil programming language and the Vortex optimizing compiler are available via anonymous `ftp` from cs.washington.edu:/pub/chambers and via the World Wide Web URL http://www.cs.washington.edu/research/projects/cecil.

References

[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, Oregon, October 1994.

[Chambers & Ungar 89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for Self, A Dynamically-Typed Object-Oriented Programming Language. *SIGPLAN Notices*, 24(7):146–160, July 1989. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.

[Chambers & Ungar 91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *Proceedings OOPSLA '91*, pages 1–15, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[Chambers 92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, March 1992.

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

[Chambers et al. 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF – a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings OOPSLA '89*, pages 49–70, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[Chambers et al. 95] Craig Chambers, Jeffrey Dean, and David Grove. A Framework for Selective Recompilation in the Presence of Complex Intermodule Dependencies. In *17th International Conference on Software Engineering*, Seattle, WA, April 1995.

[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*, pages 408–431, Bologna, Italy, July 1994. Springer-Verlag.

[Cooper et al. 92] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. Procedure Cloning. In *Proceedings of 1992 IEEE International Conference on Computer Languages*, pages 96–105, Oakland, CA, April 1992.

[Dean & Chambers 94] Jeffrey Dean and Craig Chambers. Towards Better Inlining Decisions Using Inlining Trials. In *Proceedings of the ACM Conference on LISP and Functional Programming '94*, pages 273–282, Orlando, FL, June 1994.

[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.

[Garrett et al. 94] Charlie Garrett, Jeffrey Dean, David Grove, and Craig Chambers. Measurement and Application of Dynamic Receiver Class Distributions. Technical Report UW-CS 94-03-05, University of Washington, March 1994.

[Grove & Torczon 93] Dan Grove and Linda Torczon. Interprocedural Constant Propagation: A Study of Jump Function Implementations. *SIGPLAN Notices*, 28(6):90–99, June 1993. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[Hölzle & Ungar 94] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. *SIGPLAN Notices*, 29(6):326–336, June 1994. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.

[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 1991. Springer-Verlag.

[Ingalls 86] Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proceedings OOPSLA '86*, pages 347–349, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[Jones et al. 93] Neil D. Jones, Carstein K. Gomarde, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, NY, 1993.

[Katz & Weise 92] M. Katz and D. Weise. Towards a New Perspective on Partial Evaluation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation '92*, pages 29–36. Yale University, 1992.

[Kiczales & Rodriguez 89] Gregor Kiczales and Luis Rodriguez. Efficient Method Dispatch in PCL. Technical Report SSL 89-95, Xerox PARC Systems Sciences Laboratory, 1989.

[Kilian 88] Michael F. Kilian. Why Trellis/Owl Runs Fast. Unpublished manuscript, March 1988.

[Lea 90] Doug Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, San Francisco, CA, April 1990.

[Lim & Stolcke 91] Chu-Cheow Lim and Andreas Stolcke. Sather Language Design and Performance Evaluation. Technical Report TR 91-034, International Computer Science Institute, May 1991.

[Palsberg & Schwartzbach 91] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings OOPSLA '91*, pages 146–161, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[Pande & Ryder 94] Hemant D. Pande and Barbara G. Ryder. Static Type Determination for C++. In *Proceedings of Sixth USENIX C++ Technical Conference*, 1994.

[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, Oregon, October 1994.

[Ruf & Weise 91] E. Ruf and D. Weise. Using Types to Avoid Redundant Specialization. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation '91*, pages 321–333. ACM, 1991.