Identifying Profitable Specialization in Object-Oriented Languages

Jeffrey Dean, Craig Chambers, and David Grove

Department of Computer Science and Engineering
University of Washington

Abstract

The performance of object-oriented languages can be greatly improved if methods can be specialized for particular classes of arguments. Such specialization can provide the compiler with enough class information about the receivers of messages within the specialized routine to enable these messages to be statically-bound to their target methods and subsequently inlined. We present an algorithm for automatically determining which methods are most profitable to specialize for which argument classes. This algorithm improves on previous automatic techniques by avoiding the twin problems of over- and underspecialization and by being suitable for specializing programs that use multi-methods.

1 Introduction

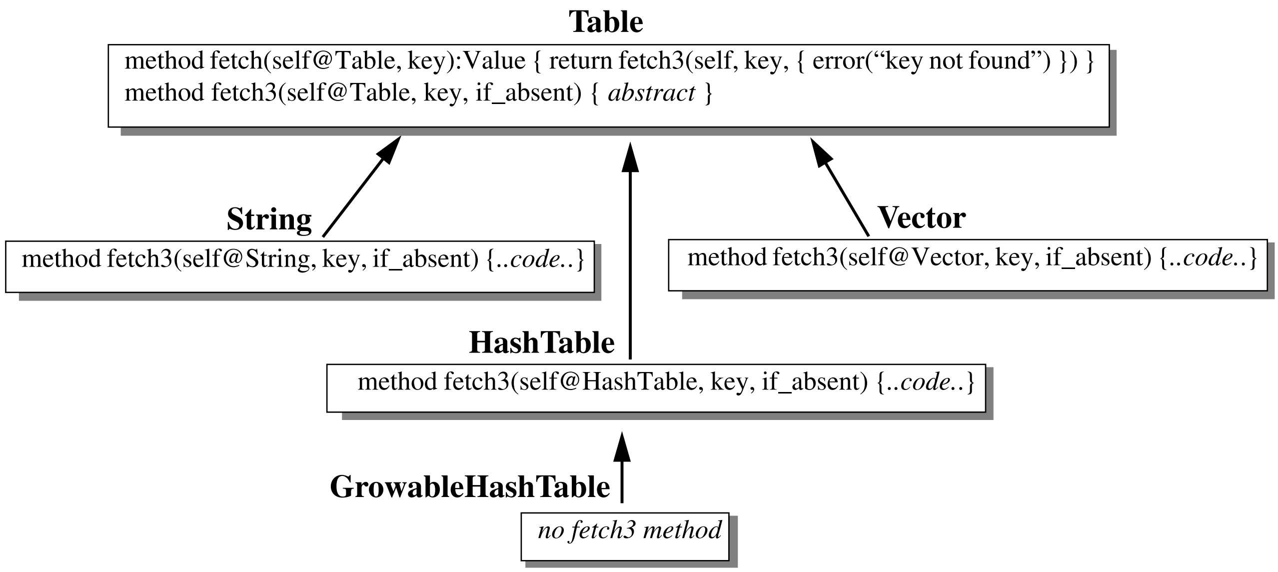
Object-oriented languages are useful for implementing abstractions and structuring programs to be more extensible and maintainable. One of the key reasons is dynamic binding (also known as message passing). Unfortunately, dynamic binding is slow, compared with a simple procedure call. In relatively pure object-oriented languages, such as Smalltalk [Goldberg & Robson 83], Eiffel [Meyer 92], Trellis [Schaffert *et al.* 86], SELF [Ungar & Smith 87], and Cecil [Chambers 92b], dynamic binding occurs very frequently and consequently its impact on performance is severe. For example, an efficient implementation of Smalltalk-80 runs a suite of small benchmarks 5 to 10 times more slowly than does optimized C, in large part due to the overhead of dynamic binding [Chambers & Ungar 91]. Even for hybrid languages that promote relatively coarse-grained use of dynamic binding, such as C++ [Stroustrup 91], Modula-3 [Nelson 91], and CLOS [Bobrow *et al.* 88], dynamic binding can become a performance bottleneck if the programming style in use encourages heavier use of object-oriented features.

Program performance can be improved if the target of a dynamic message send can be determined at compile-time. By doing so, the cost of determining the target method at runtime is eliminated, and these statically bound call sites also become amenable to further optimizations such as inlining. In order to statically bind a call site, however, the compiler requires static information about the possible classes of the receiver of a message. One way of obtaining this class information is to produce a version of a method specialized for a subset of its possible argument classes. This provides the compiler with additional information about the classes of the method's formal arguments, permitting messages sent to the formals to be statically bound and potentially inlined. This can greatly improve performance. For example, SELF programs run from 1.5 to 5 times faster as a result of specialization [Chambers 92a].

In this paper we present an algorithm that identifies automatically those points in the program where specialization is profitable using call graphs derived from dynamic profiles of the program. Section 2 provides an example motivating the benefits of specialization. Section 3 discusses previous work and identifies several shortcomings which are addressed by our algorithm. Section 4 presents our algorithm. Section 5 describes our current implementation status. Section 6 discusses related work, and section 7 offers our conclusions.

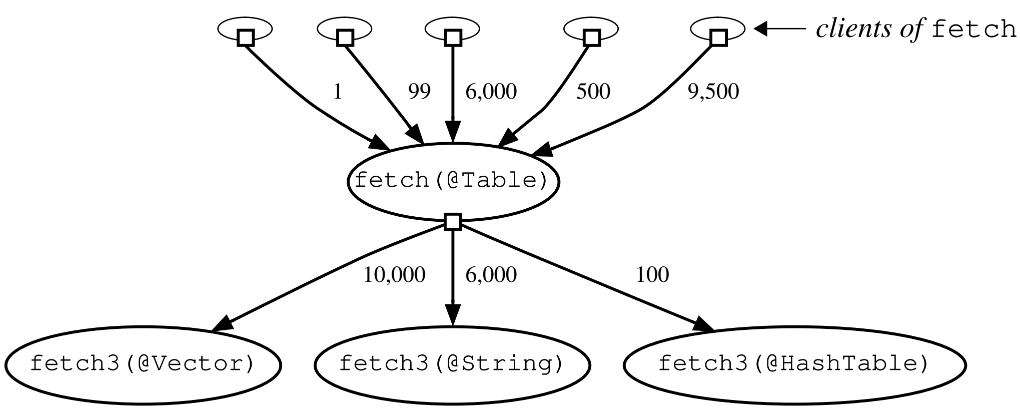
2 A Motivating Example

To illustrate the issues surrounding specialization of methods in object-oriented languages, we present an example of a keyed table class hierarchy and the basic table lookup method, `fetch`. In the inheritance graph below, class names are shown in boldface type, inheritance relations are indicated by arrows, and methods defined for a class are shown in the boxes:



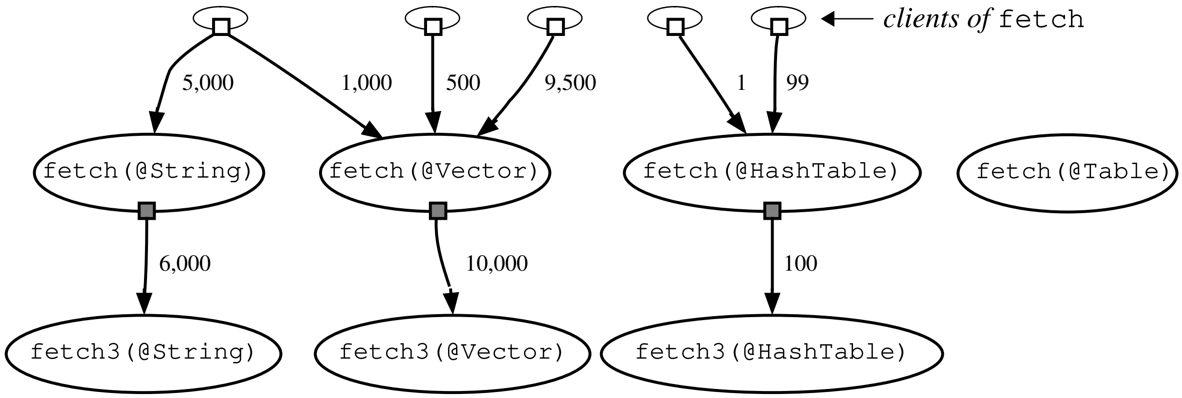
The `fetch` method is the primary interface to tables. This method is defined in the `Table` abstract class and is inherited by all data structures that support a table-like interface. The `fetch` method in turn sends the `fetch3` message to itself with a default error-handling closure (enclosed in braces) as an extra argument. Each concrete implementation of a table defines its own `fetch3` method, except for `GrowableHashTable` which inherits the `fetch3` method from its superclass, `HashTable`.

Call graphs are a useful framework with which to describe the effects of specialization. A call graph represents the calling structure of a program: nodes correspond to methods and edges represent calls from one method to another. For the call graphs in this paper, call sites are indicated with small squares, which are white □ if the call site is dynamically bound, and are shaded ■ if the call site is statically bound. Edges have a weight associated with them, indicating the number of times the edge was traversed during execution. The following call graph illustrates a program fragment where five callers send the `fetch` message:



The `fetch` method sends the `fetch3` message to `self`, which is dynamically dispatched since three different `fetch3` methods can be invoked from this one call site.

By specializing the `fetch` method for different subsets of its potential receiver classes, the compiler can obtain more precise static information in the specialized versions about the class of the receiver of the `fetch3` message, enabling the message to be statically bound. The call graph below shows one way of specializing that statically binds all of the calls to `fetch3` routines:



By producing a specialized version of `fetch` for each of the three main kinds of tables and thereby statically-binding the calls to `fetch3`, 16,100 (10,000 + 6,000 + 100) dynamic dispatches were eliminated. These statically-bound `fetch3` call sites are now amenable to inlining to further improve performance.

Whether or not it is profitable to perform these specializations depends on the desired trade-off between run-time performance improvement and compile time and code space costs. In this example, the system might elect to not specialize `fetch` for `HashTable`'s, since there isn't much benefit for specializing it (a savings of only 100 dynamic dispatches) but there is a significant cost in terms of increased compiled code space and compilation time. Similarly, the system needn't produce a specialized version of `fetch` for `GrowableHashTable`'s, since `fetch` never called any routines (directly or indirectly) where the knowledge that the receiver was a `GrowableHashTable` would have provided any additional benefit over knowing that it inherited from `HashTable`.

We used `fetch` as an example to illustrate the tradeoffs in specialization. Of course, for such a simple method, inline-expansion would be a more effective optimization; inlining achieves the effect of specializing the called routine in the context of all the data flow information available at the call site, and also eliminate the call-return overhead associated with out-of-line specialization. However, only small routines can be so aggressively specialized without suffering from a severe explosion in code space and compilation time. There remains a large class of methods that are too large to inline-expand but still benefit from specialization. For example, hash table lookups are usually too large to inline effectively but they can benefit significantly from specialization.

3 Previous Work in Object-Oriented Languages

Method specialization has been incorporated into several existing implementations of object-oriented languages, including SELF [Chambers & Ungar 91], Trellis [Kilian 88], and Sather [Lim & Stolcke 91]. Each of these systems has used a fairly simple-minded strategy to determine when to specialize a method: a method is always specialized on the exact class of the receiver, for all receiver classes, and never on anything else. While this approach is simple to implement and enables many sends to `self` to be statically bound and potentially inlined, it can lead to *overspecialization* and/or *underspecialization*.

- In some cases, specializing a routine for a single receiver class is overly precise: little or no additional benefit is obtained over specializing for a group of several receiver classes. In the example above, producing a version of `fetch` specialized for `GrowableHashTable`'s provides no additional benefit over a version of `fetch` specialized for `HashTable`'s: the generated code for these two specializations would be the same. The Trellis implementation includes a pass after specialization that shares code among specializations that turn out to have identical machine code after specialization, alleviating the code space problem in some circumstances but still consuming compilation time.

  For large programs with deep inheritance hierarchies, producing a specialized version of every routine for every potential receiver class can lead to significant code explosion. The SELF implementation often avoids this problem by producing specializations lazily at run-time, as part of its general strategy of dynamic compilation. If a method is never invoked for a particular receiver class, then no compiled method will be produced. However, this strategy can still lead to 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 a particular receiver class, as was the case with the `fetch3` message sent to `HashTable` objects. Moreover, dynamic compilation may not be a suitable framework for some programming systems.

- By never specializing on an argument other than the receiver, these systems can miss some opportunities to boost performance. Although message sends to the receiver comprise a significant fraction of the message sends within most methods, message sends to arguments other than the receiver are also common, and by not specializing on the arguments the compiler may lack the precise class information that is crucial to statically binding these messages. However, the equally simplistic strategy of specializing on all arguments can lead to even worse code explosion.

  Specializing on multiple arguments seems particularly appropriate for object-oriented languages incorporating multi-methods, such as CLOS and Cecil. Multi-methods generalize standard singly-dispatched methods by allowing dynamic binding to be based on the dynamic class of any subset of the message's arguments, not just the receiver argument. In such a situation, there is no clearly-defined single receiver argument that is appropriate for specialization, and consequently more intelligent decisions are needed to determine the combinations of argument classes that are worth specializing.

The algorithm presented in this paper addresses the problems of over- and underspecialization. Specializations of methods are made only as specific as is necessary to statically bind important (high weight) calls, and situations where it is profitable to specialize a routine on arguments other than its receiver are identified. We are developing the algorithm in the context of a language based on multi-methods.

Lea has explored the potential benefits of applying specialization in the C++ language [Lea 90]. For a simple matrix multiplication benchmark where the index function was dynamically bound, Lea calculated a potential speedup of around a factor 10 if the multiply routine were specialized to the particular matrix representations, thereby statically-binding the index call. Lea proposed user annotations to guide where specialization was to be applied, but did not implement his proposal.

4 A Specialization Algorithm

This section explains our algorithm for determining where specialization is profitable. The next subsection discusses how we measure profitability. Subsection 4.2 presents the basic algorithm, with subsection 4.3 explaining how the basic algorithm is extended to cope with closures. Subsection 4.4 discusses how the algorithm is adapted to singly- and multiply-dispatched systems.

4.1 Profitability

Our algorithm attempts to balance the costs and benefits of specialization to improve program performance without excessive space and compile time costs. We measure the benefit of specialization by the dynamic number of dynamically-bound message sends which are turned into statically-bound calls due to the additional information provided by specializing. The cost of specialization is measured by the increase in code space introduced by the specialized copies of routines. Our current implementation estimates profitability by examining the cost-benefit ratio of specialization. Based upon the desired balance between code space and compile time costs, and compiled program performance, we can select a maximum value for the cost-benefit ratio that will be considered to be profitable.

These costs and benefits do not fully account for the effects of specialization. In particular, they do not account for the performance improvement due to post-specialization inlining and subsequent optimization. Our profitability metric can be extended to take these secondary effects into account by incorporating a persistent database that records the costs and benefits of inlining [Dean & Chambers 94]. This information could then be combined with other cost-benefit information to improve the accuracy of the profitability estimation.

Our algorithm identifies profitable places to specialize by examining the program's call graph. Our current implementation uses a call graph constructed using profile data from previous runs of the program.* Edge weights in the call graph help to identify portions of the call graph which are important enough to warrant specialization. Other compilation systems have

---

* The Cecil implementation uses polymorphic inline caches (PICs) to speed dynamic dispatch [Hölzle *et al.* 91]. A PIC is a call-site-specific method lookup cache, mapping argument classes for a message send to the routine which should be invoked. When a Cecil program terminates, its PICs contain a complete profile of the classes of arguments which appeared during program execution for each send location in the program, and this provides sufficient information to reconstruct the call graph. We augment the PICs to keep a counter of the number of times each argument class occurred, to provide weights on each edge in the call graph.

exploited dynamic profiles to guide the application of optimizations, such as the Impact-C profile-guided inliner [Chang *et al.* 92].

A call graph constructed from profile data is not guaranteed to be conservative; a future program execution might traverse an edge not found in the call graph. Fortunately, our algorithm does not require that the call graph be conservative, since producing or not producing a specialization does not affect program correctness. In the absence of other conservative information, we must however preserve the original unspecialized routine to catch cases where an edge is traversed with a receiver class not encountered during the profile-gathering program execution.

4.2 The Basic Algorithm

Two characteristics of call graph edges are important for our algorithm:

- A *pass-through edge* is an edge in which one or more of the formals of the caller are passed through as actual arguments to the call site that is the source of the edge. All of the edges between the `fetch` method and the `fetch3` methods in the example call graphs are pass-through edges, because the send of the `fetch3` message passes through the `self` and `key` formals of the `fetch` method to the `fetch3` call site. Conversely, a *non pass-through edge* is one in which none of the caller's formals are passed through as parameters of the message send.
- A *statically-bound edge* is an edge where the class information available in the caller is sufficient to uniquely determine the callee at compile-time. A *dynamically-bound edge* requires run-time dispatching to determine the callee.

A *dynamically-bound*, *pass-through edge* is one which might benefit from specialization, since knowing additional class information about the formals of the caller might allow the call to be statically-bound. In some situations, a *dynamically-bound, non-pass through edge* might benefit from specialization. For example, a formal might be copied into a local before it is used as an argument to a call. Our framework could be extended straightforwardly to account for more complex relationships between formals and actuals. Grove and Torczon compare the utility of several "jump functions" representing such relationships in the context of interprocedural constant propagation [Grove & Torczon 93].

The algorithm is presented below interspersed with discussion about its various pieces. The presentation is in the style of a literate program [Knuth 92]. When helpful, call graph diagrams illustrating the various situations which occur are presented. In these diagrams, *pass-through* edges are indicated as solid black lines, and *non pass-through* edges are shown as gray lines. We continue our convention of representing *statically-bound* calls with solid square and *dynamically-bound* calls with white squares.

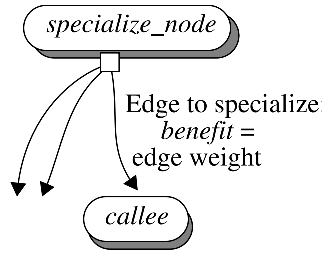
---

specialize_program is the top level routine of the algorithm. It loops, looking for edges which might benefit from specialization, until a space budget allotted for specialization has been exceeded. It examines edges in decreasing order

---

specialize_program(*budget*)
    while *cost* < *budget* **do**
        Let *e* := highest weight edge in call graph that is dynamically-bound, pass-through and is not visited
        cost := *cost* + ***specialize***(*e.caller*, *e.callee.specializers*, *e.weight*)
        e.visited := **true**
    end
end ***specialize_program***

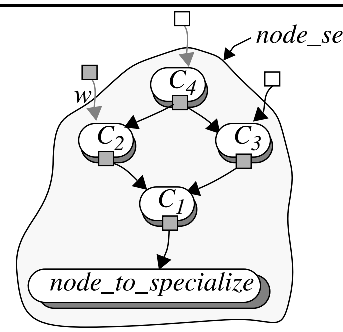
specialize is called for each dynamically-bound edge that we are attempting to statically-bind through specialization. It is given three parameters: the caller of the edge we're trying to statically bind (*specialize_node*, which is the node we will potentially specialize), the *desired_info*, which is a description of the class information required to make the edge statically bound, and the *benefit*, which is the number of dynamically-bound sends which will be turned into statically bound sends if the edge can be statically bound (simply the weight on the edge between *specialize_node* and *callee*). There may be other routines called from this call site other than the callee (since the call site is, by definition, dynamically bound): these are shown with additional arrows.



The routine returns the total cost of all nodes which are to be specialized.

specialize(*specialize_node*, *desired_info*, *benefit*)
    nodes_to_specialize := ∅
    collect_nodes(*specialize_node*, *desired_info, benefit*, *nodes_to_specialize*)
    cost := sum of space cost for each node in *nodes_to_specialize*
    if *specialization_is_profitable*(*cost*, *benefit*) **then**
        output directives to specialize members of *nodes_to_specialize* for *desired_info*
    else
        cost := 0
    end
    return *cost*
end*specialize*

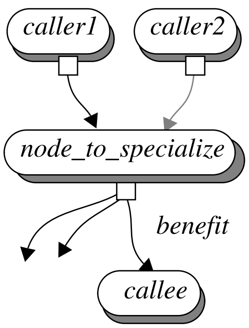
collect_nodes is called to determine the connected subgraph of nodes which reach *node_to_specialize* through statically-bound, pass-through edges. This set is of particular interest because such nodes should be specialized as a unit. If they were not specialized together, then calls which were statically-bound will be turned into dynamically-bound calls unnecessarily. By specializing as a unit, we avoid introducing new dynamically-bound calls which will have to be specialized back into statically-bound calls.



The routine computes *node_set*, the set of nodes to be specialized. It also adjusts the *benefit* value downward by the number of statically-bound non-pass-through calls which may be turned into dynamically-bound calls.

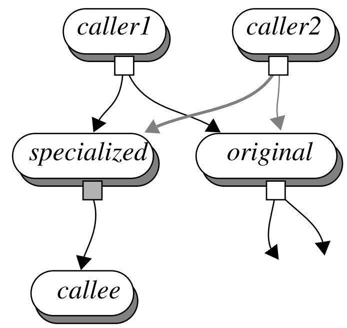
collect_nodes(*node_to_specialize*, *desired_info*, **var** *benefit*, **var** *node_set*)
    /* Return immediately if node already visited (recursion) */
    if *node_to_specialize* ∈ *node_set* **then**
        return
    end
    node_set := *node_set* ∪ { *node_to_specialize* }
    foreach incoming edge *e* of *node_to_specialize* **do**
        /* We want to take different action, depending on which of the four varieties of edge this is: */

Before



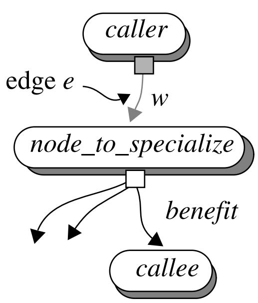
For the case of a dynamically-bound edge, specialization incurs no reduction in benefit, since the call to *node_to_specialize* is already dynamically dispatched. This is true regardless of whether the edge entering *node_to_specialize* is a *pass_through* edge (*caller1*) or not (*caller2*). If the newly introduced edge from *caller1* to *specialized* has sufficient weight, then *caller1* may potentially be split during a subsequent iteration of the main loop. This can cause a rippling effect upwards as a whole call chain is specialized.

After



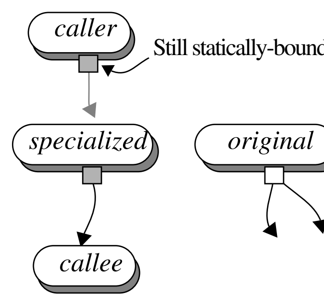
if is_dynamically_bound(*e*) **then**
  nothing extra to do for dynamically bound incoming edges

Before



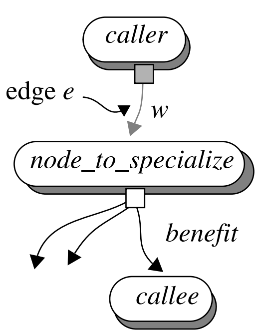
For a statically-bound edge, we first check whether the call site has enough static information to remain statically bound after specialization, either to the *specialized* or to the *original* method. If this is the case, then the full benefit of specialization remains, and no specialization of the caller is needed.

After



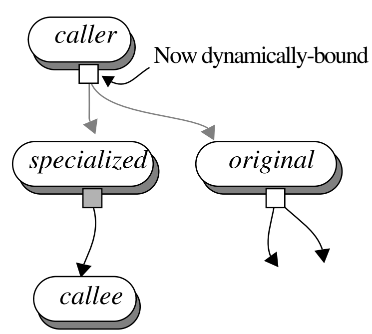
else /* is_statically_bound(*e*) */
  if ***has_desired_info***(*e, desired_info*)) **then**
    already has the desired info; will still be statically bound after specialization without cost
  else
    /* Need to consider effect of specialization on statically-bound caller */

Before

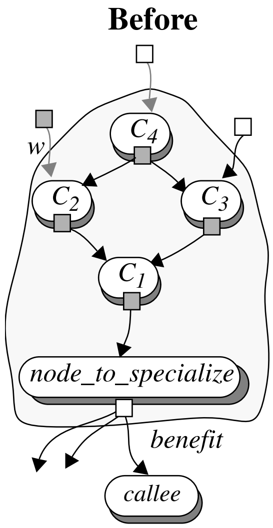


For a statically-bound edge that does not pass-through any formals, the benefit of specialization is tempered by the fact that the statically-bound call from *caller* to *specialize* may become dynamically-bound after specialization, if the static information available in *caller* is insufficient to statically determine whether the *specialized* or *original* method will be called. To be conservative, the algorithm reduces the benefit of specialization (*benefit)* by *w*, the weight of the edge from *caller* to *specialize*. Specialization is only a win if *benefit* > *w* (if *benefit* *w*, then specialization only moves dynamic calls from one location to another).

After

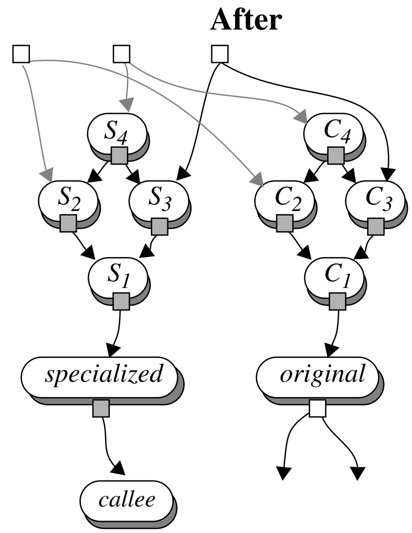


if not(is_pass_through(*e*)) **then**
  benefit := *benefit* - *e*.weight



We attempt to specialize all statically-bound, pass-through callers of the *node_to_specialize* routine as a whole. This avoids converting statically-bound pass-through calls into dynamically-bound calls unnecessarily. To find the subgraph that reaches specialize through statically-bound, pass-through calls, we recursively call **collect_nodes** for the caller of the edge to add the reachable nodes to the *node_set*.

In the diagram at the left, the computed *node_set* is {*node_to_specialize*, C_1, C_2, C_3, C_4}. After specialization, specialized versions for each of these nodes would be created.




```
                else /* is_pass_through(e) */
                    /* Recur on this edge, collecting nodes */
                    collect_nodes(e.caller, desired_info, benefit, node_set)
                end
            end
        end
    end
end collect_nodes
```

specialization_is_profitable makes a final yes or no decision about a potential specialization, based on the cost and benefit. The exact nature of the heuristics depends on the desired trade-offs between code space and compile time vs. run-time performance.

```
specialization_is_profitable(cost, benefit)
    return true if cost is acceptable, given the benefit; false otherwise
endspecialization_is_profitable
```

has_desired_info accepts a call graph edge, *e*, and a description of class information, *desired_info*, and returns true if *e*'s caller is guaranteed to have the *desired_info* available statically.

```
has_desired_info(e, desired_info)
    return true if e's caller has desired_info available statically; false otherwise
end has_desired_info
```

The output of this algorithm is a list of specialization directives. These directives are first processed to merge directives to specialize the same routine in non-interfering ways. If a routine contains multiple call sites, each to a different formal of the routine, each call site can generate distinct specialization directives. These directives are merged where they do not conflict (specialize the same formal for different types) to reduce the number of specializations computed. (We are investigating algorithms to generate specialization directives that more directly account for multiple specializable call sites within a single routine.) Once the set of directives is computed, our system obeys the directives to generate the specializations.

As written, the algorithm has worst-case time complexity $O(E^2)$, where E is the number of edges in the call graph. By segregating the various types of edges and only visiting dynamically-bound pass-through edges in the specialize_program loop and statically-bound edges in the collect_nodes loop, the algorithm reduces to $O(DP * SP + SN)$, where DP is the number of dynamically-bound pass-through edges, SP is the number of statically-bound pass-through edges, and SN is the number of statically-bound non-pass-through edges. The product $DP * SP$ is significant only when a large number of statically-bound pass-through edges are repeatedly collected as part of processing a large number of dynamically-bound edges, which we consider a rare occurrence. Consequently, we expect this algorithm to take time roughly linear in the number of edges in practice, and our initial experimental results confirm that identifying specializations is quick.

4.3 Closures

In the presence of first-class, lexically-nested function objects, such as blocks in Smalltalk and SELF and closures in Cecil, a message can be sent to a formal not of the sending routine but of a lexically-enclosing routine. We extend our specialization algorithm to support this case by computing the node_to_specialize not as the caller of the edge being specialized but as the routine that declares the formal that is being passed through to the call site. In many situations this will be the same as the caller of the edge being specialized, but in the presence of closures this can be a routine that lexically encloses the caller of the edge.

4.4 Specialization in Single- and Multiple-Dispatching Systems

Our specialization algorithm allows any subset of a routine's arguments to be considered a candidate for specialization. The subset selected is determined from the messages sent by the routine and the arguments of those messages that are subjected to run-time type tests to determine the target method. In the presence of multi-methods, several of the arguments of a particular dynamically-bound call site might be subject to run-time type tests, in which case the desired_info manipulated by the algorithm will be a set of <variable, desired class> pairs.

To implement specialization, our implementation generates new methods whose formals are restricted to apply to particular subsets of classes. This approach exploits our environments multi-method dispatching infrastructure, using multi-method dispatching to test when a particular specialized version is appropriate. This infrastructure may not be present in a singly-dispatched system. To exploit our algorithm in such a system, we recommend that the basic run-time system of the language be augmented to support selecting method implementations based on the classes of several arguments. Alternatively, versions of a routine specialized on arguments could be limited to being invoked only from call sites that possess statically the required class information, with other cases being caught by the original unspecialized routine.

5 Implementation Results

We have implemented the described algorithm in the context of the Cecil compiler. Our current system constructs a program's call graph derived from profiles of the program, generates specialization directives using the algorithm described in this paper, and produces specialized versions of routines based on these specialization directives. As one measure of the effectiveness of the algorithm, the specialization directives produced by our algorithm agree with our intuitive sense about what specializations are most profitable. For a more empirical assessment, we examined the cost of our implementation of the algorithm, measured by the number of generated specializations, and its benefits, measured in runtime performance of the program after specialization. In order to provide a realistic benchmark, all of our measurements are based upon the specializations produced by the algorithm when it was run on the Cecil compiler program, a 30,000-line Cecil program. The speed of executing the algorithm itself is good, taking a few seconds for small programs and under 5 minutes for computing specializations for the large Cecil compiler program.

Table 1 compares the number of specializations that would be generated for this program by a static approach (as used in Sather and Trellis), by a dynamic compilation-based approach (as used in SELF), and by our selective specialization algorithm, for both singly- and multiply-dispatched systems. The static, per-class column reflects the number of specialized methods that would be generated if each source method were specialized for each of the possible classes of its first argument, for the single-dispatching row, or for all possible combinations of subclasses of the dispatched arguments of the method, for the multiple-dispatching row. To model the dynamic compilation-based strategy, we used profile information to determine which combinations of arguments occurred during a run of the program. We then restricted the static strategy to only generate specializations for receivers (or combinations of dispatched arguments) that occurred during the profiled execution. The "dynamic methods called" column indicates that only around half of the statically-defined methods were actually invoked during the profiled execution. The final two columns report the number of specializations generated by our selective specialization algorithm under two different heuristic settings: a "normal" setting and a setting that is more aggressive in deciding the specializations that are profitable. The numbers in these columns include the static number of methods in the program (because our algorithm always preserves the unspecialized version of a method), plus the number of additional specialized methods generated by our algorithm, to give the total number of methods in the specialized program.

Table 1: Generated Specializations

	Static Source Methods	Static, Per-Class	Dynamic Methods Called	Dynamic, Per-Class	Selective Specialization	Aggressive Selective Specialization	Dynamic Selective Specialization
Single Dispatch	5812	16,917	2883	5299	5812+233=6045	5812+526=6338	2883+233=3116
Multiple Dispatch		172,824		6226	5812+247=6059	5812+967=6779	2883+247=3130

As the above numbers indicate, statically specializing a method for each possible receiver class is unattractive for large programs with deep inheritance hierarchies and its extension to multiply-dispatched systems is completely infeasible. A dynamic approach is more attractive, but requires adopting a dynamic compilation model, which is impractical in many environments. Our selective algorithm produces many fewer specializations than the static, per-class specialization scheme and is on a par with the dynamic specialization strategy. This is remarkable because the dynamic figures are for a particular program execution that only exercises only 2883 out of 5812 methods in the program. The number of specialized methods for the dynamic compilation approach would increase significantly as more of the program was exercised, while our approach would focus specialization effort on those parts of the program where specialization really provides benefits. Our algorithm could be adapted to a dynamic compilation-based environment by using precomputed specialization directives to decide if and how to specialize a method when it is first invoked at run-time. This would lead to significantly fewer specializations than the simple dynamic per-class strategy, as the last column indicates.

To measure the bottom-line effects of the algorithm, we compiled the Cecil compiler program using standard optimizations, standard optimizations plus specialization, and standard optimizations plus aggressive specialization. Our standard optimizations include intraprocedural concrete type analysis, inlining, dead code elimination (to optimize away unneeded closure creations), and hard-wired type prediction for a small number of common messages such as `if` and `+`. We then measured the time taken by each of the three compiler executables to compile a small suite of Cecil programs. Table 2 shows the relative execution time and compiled code space for the three versions of the compiler program.

Table 2: Runtime Performance and Code Space Usage

	Standard Optimizations	plus Selective Specialization	plus Aggressive Specialization
Execution time	1.00	0.67	0.66
Compiled code size	1.00	1.05	1.27

Our specialization algorithm yields significant runtime improvement with only small increases in code space cost (and the corresponding compile time required). Since the aggressively specialized program shows little improvement over the specialized program, we conclude that our algorithm is working well at selecting profitable methods to specialize.

6 Related Work

The implementations of SELF [Chambers & Ungar 91], Trellis [Kilian 88], and Sather [Lim & Stolcke 91] use specialization to provide the compiler with additional information about the classes of arguments to a routine, allowing many message sends within the routine 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. As discussed in section 3, this can lead to both overspecialization and underspecialization. Our approach is more precise because it identifies sets of receiver classes which enable static-binding of message sends send to the receiver (thereby avoiding overspecialization), and because it allows specialization on arguments other than just the receiver of a message (preventing underspecialization for arguments).

The techniques described in this paper focus on using specialization to convert dynamically-bound message sends into statically-bound calls. However, many dynamically-bound message sends still exist in the residual program. Other work focuses on reducing the overhead of dynamic dispatching in object-oriented programs, by partially evaluating the routines to perform method lookup with respect to the inheritance hierarchy of the program being executed [Khoo & Sundaresh 91, Harnett & Montenyohl 92]. We rely on standard caching techniques to reduce the overhead of dynamic dispatching [Krasner 83, Hölzle *et al.* 91].

Cooper, Hall, and Kennedy present a general framework for identifying when creating multiple, specialized copies of a procedure can provide additional information for solving dataflow optimization problems [Cooper *et al.* 92]. Their approach begins with the program's call graph, makes a forwards 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 equivalent cloning vectors and identifying specializations. Their framework applies to any forward dataflow analysis problem, including constant propagation and concrete type analysis. Our work differs from this framework 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. 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 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 guide its search for profitable specializations.

Ruf, Katz, and Weise [Ruf & Weise 91, Katz & Weise 92] also address the problem of avoiding overspecialization. Their work seeks to identify when two specializations end up leading to the same code. Ruf identifies the subset of information 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, we exploit dynamic profile information to be more selective in where to apply specialization, and our algorithm can choose to avoid specialization that does not pay for itself, even if some optimizations could be performed in the specialized routine.

7 Conclusions

We have presented an algorithm for determining which methods in an object-oriented language should be specialized for which set of argument classes. Our algorithm uses weighted call graphs derived from dynamic profiles of the program to determine those parts of the program with high execution. Our algorithm strives to balance the benefits of specialization against its costs, improving on previous automatic specialization algorithms by avoiding both overspecialization and underspecialization. As a consequence of its more judicious application of specialization, our algorithm is appropriate for specializing on multiple arguments of a method and coping with multi-methods. Initial results from our implementation of the algorithm are encouraging: a 50% speed improvement for a substantial application program with only a 5% space increase. The algorithm and its approach of exploiting the program's weighted call graph could be adapted for other kinds of languages to help select the routines that are most profitable to specialize.

Acknowledgments

This research is supported in part by a National Science Foundation Research Initiation Award (contract number CCR-9210990) and several gifts from Sun Microsystems, Inc. The comments from the anonymous referees improved the presentation of this paper. We thank Stephen North and Eleftherios Koutsofios of AT&T Bell Laboratories for producing `dot`, a program for automatic graph layout; `dot` has been invaluable in visualizing large call graphs.

References

[Bobrow *et al.* 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon. Common Lisp Object System Specification X3J13. In *SIGPLAN Notices 23(Special Issue)*, September, 1988.

[Chambers & Ungar 91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Conference Proceedings*, pp. 1-15, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(10)*, October, 1991.

[Chambers 92a] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph.D. thesis, Department of Computer Science, Stanford University, report STAN-CS-92-1420, March, 1992.

[Chambers 92b] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *ECOOP '92 Conference Proceedings*, pp. 33-56, Utrecht, the Netherlands, June/July, 1992. Published as *Lecture Notes in Computer Science 615*, Springer-Verlag, Berlin, 1992.

[Chang *et al.* 92] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-Guided Automatic Inline Expansion for C Programs. In *Software—Practice and Experience 22(5)*, pp. 349-369, May, 1992.

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

[Dean & Chambers 94] Jeffrey Dean and Craig Chambers. Toward Better Inlining Decisions Using Inlining Trials. To appear in *Proceedings of the ACM Symposium on Lisp and Functional Programming Languages*, Orlando, FL, June, 1994. An earlier version appears as technical report 93-05-05, Department of Computer Science and Engineering, University of Washington, May, 1993.

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

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

[Harnett & Montenyohl 92] Sheila Harnett and Margaret Montenyohl. Towards Efficient Compilation of a Dynamic Object-Oriented Language. In *Proceedings of the 1992 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 82-89, San Francisco, CA, 1992. Published as Yale University Department of Computer Science Technical Report YALEU/DCS/RR-909.

[Hölzle *et al.* 91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches. In *ECOOP '91 Conference Proceedings*, pp. 21-38, Geneva, Switzerland, July, 1991.

[Katz & Weise 92] Morry Katz and Daniel Weise. Towards a New Perspective on Partial Evaluation. In *Proceedings of the 1992 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 29-37, San Francisco, CA, 1992. Published as Yale University Department of Computer Science Technical Report YALEU/DCS/RR-909.

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

[Khoo & Sundaresh 91] Siau Cheng Khoo and R. S. Sundaresh. Compiling Inheritance using Partial Evaluation. In *Proceedings of the 1991 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 211-222, New Haven, CT, 1991.

[Knuth 92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, Lecture Notes Series, 1992.

[Krasner 83] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

[Lea 90] Douglas Lea. Customization in C++. In *Proceedings of the 1990 Usenix C++ Conference*, pp. 301-314, 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.

[Meyer 92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

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

[Ruf & Weise 91] Erik Ruf and Daniel Weise. Using Types to Avoid Redundant Specialization. In *Proceedings of the PEPM '91 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*s, pp. 321-333, New Haven, CT, June, 1991. Published as *SIGPLAN Notices 26(9)*, September, 1991.

[Schaffert *et al.* 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*, pp. 9-16, Portland, OR, September, 1986. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley, Reading, MA, 1991.

[Ungar & Smith 87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, FL, October, 1987. Published as *SIGPLAN Notices 22(12)*, December, 1987. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.