# Measurement and Application of Dynamic Receiver Class Distributions

Charles D. Garrett, Jeffrey Dean,
David Grove, and Craig Chambers

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195 USA

Technical Report 94-03-05
May 1994

{garrett,jdean,grove,chambers}@cs.washington.edu
(206) 685-2094; fax: (206) 543-2969

# Measurement and Application of Dynamic Receiver Class Distributions

Charles D. Garrett, Jeffrey Dean, David Grove, and Craig Chambers

Department of Computer Science and Engineering, FR-35
University of Washington

## Abstract

Dynamic binding slows down object-oriented programs. Dynamic dispatch mechanisms which work well where all receiver classes are equally likely are too pessimistic because at most call sites one receiver class predominates. We apply dynamic profile information to determine the dynamic execution frequency distributions of the classes of receivers at call sites. We show that these distributions are heavily skewed towards the most commonly occurring receiver class across several different languages. Moreover, we show that the distributions are stable across program inputs, from one version of a program to another, and even to some extent across programs that share library code. Finally, we demonstrate that significant run-time performance improvements for object-oriented programs can be gained by exploiting the information contained in dynamic receiver class distributions in a relatively simple optimizing compiler.

## 1 Introduction

Profiling is an important tool in the design and implementation of efficient and correct programs. For example, frequency data reveals hot spots where program optimization can have the greatest effect and basic block counts can expose incorrect behavior [Graham *et al.* 82]. Profiles of object-oriented languages can yield new information: the dynamic frequency of occurrence of each class of receiver at a particular call site. Such profile-based information complements static information derived by analyzing the program text. Whereas static analysis provides guarantees on the possible classes of the receiver of a message but little or no information on their relative frequency, profiles give frequency information but can make no guarantees.

In the absence of profile information about receiver class distributions, the compiler must make worst-case assumptions when compiling a potentially polymorphic message. In our environment, such dynamically-dispatched messages require anywhere from 10 to 30 machine instructions to dispatch; even in the fastest dynamic dispatching techniques, several machine instructions and loads must be executed on top of the call overhead [Rose 89]. Even worse, dynamic binding prevents interprocedural optimizations such as inlining. If the frequency of dynamically-bound calls is high, as it is in pure object-oriented languages like Smalltalk [Goldberg & Robson 83], or if a heavily object-oriented style is being used in a hybrid language such as C++ [Stroustrup 91], dynamic binding can be a significant bottleneck in the performance of the system.

As one example, consider a drawing editor program which has to scale a screenful of objects stored in a list:*

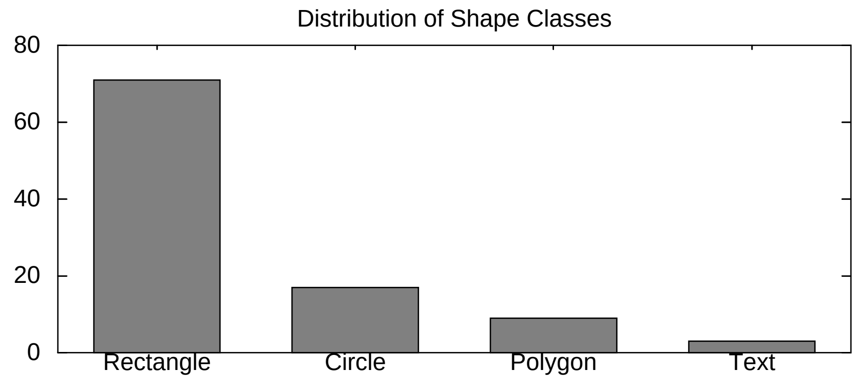


Figure 1: A sample distribution showing the receiver classes of the `scale` message. Since rectangles are much more common than other classes, the benefits of testing for rectangles will outweigh the penalty incurred for other receiver classes.

```
shapes.do(λ(s:shape){
   s.scale(dx, 1); -- stretch by dx in the x direction and 1 in the y direction
});
```

Both the `do` message and the `scale` message are potentially polymorphic: `shapes` might be one of many possible collection representations, and `s` might be one of several shape representations manipulated by the drawing editor. (We highlight the dynamically bound messages and closure creations in our examples to emphasize the slow operations.) This loop may be a performance bottleneck in the editor.

We wish to exploit receiver class distributions derived from profiles to guide optimizations in simple compilers for object-oriented languages that can improve the performance of dynamically-bound sends. Given a receiver class frequency distribution for a call site, the compiler can insert run-time tests for the most commonly occurring receiver classes. When successful, a class test branches to a copy of the send that now has a single statically-known receiver class. Such a message can be statically bound and inlined, greatly reducing the cost of the message for that receiver class. Infrequently occurring receiver classes which are not tested for at the call site will default to the dynamically-bound message. We call this technique *receiver class prediction*.

For example, if a profile of the scale method indicated that it was most frequently sent to rectangle objects, as illustrated in Figure 1, the compiler could insert a test for the receiver class before the `scale` message. This message could then be statically bound to the following method:

```
-- method to scale a rectangle
rectangle::scale(dx:num, dy:num):void {
   self.width  := self.width  * dx;
   self.height := self.height * dy;
}
```

Since this method is short, the compiler can expand it inline at the call site and then optimize it using the knowledge that `dy` at this call site is the constant 1. This produces the following code:

```
shapes.do(λ(s:shape){
```

[*] The expression `λ(s:shape){...}` creates a first-class function object (a closure) taking a shape argument, `s`. The closure is invoked by the `do` method for each element of the `shapes` collection.

```
      if s.class = rectangle then
         -- inlined version of rectangle::scale; change in height has been constant folded away
         s.width := s.width * dx;
      else
         s.scale(dx, 1);      -- send the dynamically-bound message
      end
   });
```

Furthermore, if profile information revealed that the drawing editor currently represented `shapes` as a linked list (the receiver class distribution for the `do` message was the single class `list`), the compiler could insert a run-time class check for this class, inline-expand the `do` message, and finally inline-expand the closure invocation within the `do` method, thereby eliminating the overhead of creating and invoking closures and encouraging the use of user-defined control structures. The final optimized code could look like the following:

```
   if shapes.class = list then
      -- common case: inline-expand the list::do method
      l := shapes;
      while l != nil do
         -- inline expand the closure argument to list::do
         s := l.head;

         if s.class = rectangle then
            s.width := s.width * dx;
         else
            s.scale(dx, 1);
         end

         l := l.next;
      end
   else
      -- uncommon case: send the slow do message to an unoptimized closure
      shapes.do(λ(s:shape){
         s.scale(dx, 1);
      });
   end
```

There are now no dynamically-bound sends or closure creations along the common-case path of this loop. However, code space has increased significantly as a result of these transformations. If profile information can identify the most frequently occurring call sites in the program, then the compiler can focus its efforts on just those call sites that account for the bulk of the program's running time, keeping code space costs manageable while significantly improving performance.

In this paper we investigate the feasibility of applying dynamic profile information to guide receiver class prediction. First, we study the kinds of distributions that occur in real object-oriented programs written in three distinct languages: C++ [Stroustrup 91], SELF [Ungar & Smith 87], and Cecil [Chambers 92, Chambers 93]; our results appear in section 3. This study reveals that most receiver class distributions are heavily skewed towards the most commonly occurring receiver class. Second, we study how well profiles taken from one run of a program predict the behavior of the program when run on other input, of future versions of the program, and even of other programs that share common library code; the results of this study appear in section 4. We find that profiles are quite stable at least across runs of a program and across versions of a program. Third, we measure how well the information contained in profiles can be exploited in a simple optimizing compiler to improve execution performance of object-oriented programs. As shown in section 5, we find that our Cecil programs speed up by a factor of 2 to 4 with profile-guided receiver class prediction.

In these studies, we focus on two different ways of extracting receiver class distributions from profiles. A *call site specific* distribution is derived from the profile information of a single call site and is applied to predict the receiver classes at just that call site. A *global message* distribution is derived from all the call sites in the profile that send a particular message (such as `do` or `scale`). There are several possible ways of combining the profiles of the individual call sites to compute the global message distribution. We simply sum up the dynamic count of each receiver class of each call site sending a particular message. This has the effect of skewing the message distribution towards the distributions of the most frequently occurring call sites, which makes sense given our intended application of the distribution data. Call site specific distributions are expected to be more accurate predictors of the behavior of a call site, but they can only be used for call sites that have previously been profiled. Global message distributions are more flexible because they can be applied to call sites that have not been profiled before.

## 2 Related Work

Smalltalk-80 [Deutsch & Schiffman 84] and SELF [Chambers *et al.* 89] implementations have long incorporated techniques analogous to receiver class prediction; this optimization was called *type prediction* in the SELF work. These implementations in effect incorporated a hard-wired global message distribution table for certain commonly-occurring messages such as `+` and `if`. Performance in Smalltalk and SELF improves dramatically by optimizing these sends. In less pure languages, such as C++, such operations would be built-in and not subject to the cost (or flexibility) of dynamic binding. From one vantage point, the C++ compiler has a similar message distribution table hard-wired for `+`, `if`, array indexing, etc., with the additional information that no other “receiver classes” can occur at run-time. In all these systems, because the optimizations are limited to a few classes and operations that are built-in to the system, user-defined code cannot receive similar benefits. One of the main goals of our work is to provide the benefits of run-time class testing for all code in a program.

The current SELF implementation [Hölzle *et al.* 91, Hölzle & Ungar 94] is based on adaptive compilation, where the system recompiles and optimizes parts of a program while it is running. A key innovation of this system is *type feedback*, a form of call-site-specific receiver class prediction: the run-time system gathers call-site-specific receiver class information as the program runs, and the periodic recompilations exploit this information when recompiling routines. Not only does this SELF implementation exhibit good run-time performance, it compiles quickly and exploits available profile information automatically, without user intervention. This helps to preserve the illusion that the SELF programmer is interacting with a high-functionality interpreter that happens to run much faster. A limitation of this work is that it relies on a heavyweight run-time system incorporating dynamic compilation, which may not be appropriate for all object-oriented systems. Our studies target more traditional static compilation environments. Additionally, because in our model the programmer explicitly gathers profile data for use in optimization, the programmer can try to make the profile representative. In the SELF system the automatic recompilation mechanism sometimes optimizes the program too early, say by using the profile information from the initialization phase of the program to optimize it. Similarly, our static compilation system can exploit its global view of the program and the profile data to make optimization decisions, while the SELF system is sometimes hampered by its local view and its strict limits on compile-time. Finally, our work studies receiver class distributions in several languages and analyzes the stability of profile data; the SELF work to date has reported only bottom-line performance improvements.

Calder and Grunwald [Calder & Grunwald 94] consider several ways of optimizing dynamically bound calls in C++. They measured both the fraction of calls which had only one receiver class and the fraction which were sent to the most common receiver. Their results show that the effective amount of polymorphism used by the programs they studied is far less than the potential amount. Our C++ results confirm this general result, but the C++ programs we measured are much larger and appear to be written in a more object-oriented style based on the polymorphism they exhibit.

Wall [Wall 91] investigated how well profiles predict the execution frequency of call sites, basic blocks, references to global variable references, and other kinds of behavior. He reported that profiles from actual runs of a program are better predictors than static estimates of execution frequency. We investigate the predictive power of a different kind of information, receiver class distributions, and we consider not only cross-input stability of profiles but also cross-version and cross-program stability. Fortunately for us, receiver class distributions derived from profiles appear to be very accurate.

The IMPACT C compiler [Chang *et. al.* 91, Chang *et. al.* 92] uses profiles to guide standard optimizations such as dead code elimination, common subexpression elimination, and inlining. They report improvement in the speed of C programs using one set of inputs to predict behavior on another set. Thus they demonstrate that the stability of profiles is sufficient to yield real performance benefits. We perform a similar experiment to assess the effectiveness of profiles at optimizing dynamic calls.

## 3 Receiver Class Distributions

Receiver class prediction is effective only when the receiver class test succeeds frequently. Since class tests are inserted for the most common receiver first, the payoff of receiver class prediction depends on the dynamic count of sends which go to the most common receiver. Monomorphic sends (sends which have only one receiver class at runtime) are the best to optimize in this way, but polymorphic sends where most sends go to only one or two receivers can also benefit from receiver class prediction.

Therefore, our first order of business is to measure receiver class distributions in real object-oriented programs and determine the extent to which distributions are skewed towards the more commonly-occurring receiver classes. We also wish to learn how language-dependent and programming-style-dependent receiver class distributions are. Consequently, we studied several object-oriented programs written in C++, SELF, and Cecil. C++ exemplifies hybrid object-oriented languages, SELF represents purely object-oriented languages with sophisticated implementation, and Cecil characterizes a language with multi-methods and a relatively simple implementation. Through this study we hope to understand the kinds of receiver class distributions that occur in practice, and as a fringe benefit we hope to be able to assess the impact of language on the programming style, as revealed by the receiver class distributions.

## 3.1 Methodology

Table 1 describes the benchmark programs we ran. The specific quantities measured for each language are detailed below.

**Table 1: Benchmark Programs**

| Language | Program | Size (in lines) | Inputs |
|---|---|---|---|
| C++ | `new` SELF compiler | 33,528 | Small SELF benchmarks, Cecil compiler written in SELF |
| | `sic` SELF compiler | 14,855 | Small SELF benchmarks, Cecil compiler written in SELF |
| | `doc` editor | 15,366 + library | Typing in one page of text. Randomly cutting and pasting in an existing 10 page document. |
| | `idraw` graphical editor | 6,258 + library | Drawing lots of rectangles. Exercising all possible shapes and text. |
| SELF | OOSuite benchmarks | 3,201 + library | Default for each program |
| | Cecil compiler in SELF | 13,473 + library | Towers of Hanoi |
| Cecil | Cecil compiler in Cecil | 22,700 + 5,800-line library | Towers of Hanoi and compiler test suite, with and without inlining |

To gather receiver class profiles in C++, we exploited assembly language idioms of the virtual function calling sequence generated by the `g++` compiler to record the receiver class and target method for every virtual function call in our benchmarks; details of our C++ profiling technique are in Appendix A. We could not measure receiver class distributions for non-virtual member function calls. However, since these calls are statically bound, we would not be able to optimize them any better using profile data. We also did not measure global function calls, member variable accesses, or built-in operators like +. Since we wished to study well-written C++ programs using a heavily object-oriented style, we chose two of the most polymorphic programs we could find, the `new` [Chambers & Ungar 91] and `sic` [Hölzle & Ungar 94] compilers for the SELF language distributed with SELF version 3. We profiled both compilers while they compiled a small set of SELF benchmarks and while they compiled one large SELF program. We also profiled a document preparation program, `doc`, and a graphical editor, `idraw`, both written on top of the InterViews graphics library.

To gather profiles in SELF, we used the polymorphic inline cache runtime system mechanism [Hölzle *et al.* 91] to record distinct receiver classes and invoked methods at dynamically-bound call sites. To make the data more comparable to the C++ data, we measured SELF programs under the `new` compiler with all optimizations enabled. This had the effect of statically binding and inlining many simple messages such as `+`, `if`, instance variable accessors, and closures, using sophisticated static class analysis but not adaptive compilation. The resulting dynamic calls are those that could not be optimized using these static techniques, and so they are directly amenable to profile-guided optimization. The SELF programs we measured were an object-oriented suite of benchmarks used to assess the performance of the SELF system plus a version of the Cecil compiler written in SELF.

Cecil incorporates a similar runtime system mechanism as SELF that we exploited to gather profile information about Cecil programs. However, Cecil is a multiply-dispatched language, and no single argument is the lone "receiver." The analog of receiver class prediction for a multi-method-based language tests the class of several arguments to determine the target multi-method. For each message, the system knows which arguments can affect the outcome of method lookup (not all arguments need be tested in Cecil). We therefore consider each tuple of class checks of the dispatched arguments as a separate "receiver class test" for the purposes of measuring distributions. Many messages only dispatch on one argument, and these distributions are analogous to the distributions in C++ and SELF, while other messages dispatch on several arguments and hence may have a larger number of potential argument class combinations than singly-dispatched messages. As with the SELF measurements, we wish to gather profiles for Cecil call sites that exclude simple messages such as `+`, `if`, and closure evaluations that are not likely to be present in the C++ profiles. Accordingly, we measured profiles after simple optimizations and inlining have been performed. Because our current Cecil implementation is not as sophisticated as the SELF implementation, more closures end up being created at run-time and some messages in user-defined control structures appear extremely polymorphic. To compensate, we treat all closures as being members of a single class. Finally, in Cecil some basic operations are implemented as multi-methods which would be singly-dispatched or even built-in in other languages, such as array fetch and store operations. To compensate, we treat these messages as if they dispatched only on their first argument. The end result of these modifications is to reduce the apparent polymorphism of Cecil programs to remove artifacts of our current implementation technology and standard library design.

There is only one large Cecil program worth profiling, namely the compiler itself. We ran the compiler on two different input programs, and on each program we ran the compiler both with and without inlining; inlining exercises different parts of the compiler.

## 3.2 Measurements of Degree of Polymorphism

For each language, we derived execution frequency data from the dynamic profiles and summed together call sites that exhibited the same degree of polymorphism (i.e., that had the same number of receiver classes at run-time). The results are shown in Figure 2. The height of a bar in each histogram is the dynamic percentage of sends in the benchmarks with that degree of polymorphism. The tail of each histogram (which extends to very high polymorphism, in the 30's for C++ and higher for SELF and Cecil) has been gathered into a bin labeled 20+. Moreover, each bar of the histogram, reporting execution frequency of sends with degree *N* polymorphism, is divided vertically into *N* parts to show the relative proportion of the *N* different receiver classes; the bottom portion of the bar reports the frequency with which the most common receiver class at a call site occurred.

Because C++ somewhat discourages the use of dynamic binding, we expected that C++ programs would exhibit less polymorphism than SELF and Cecil programs. This expectation turned out to be

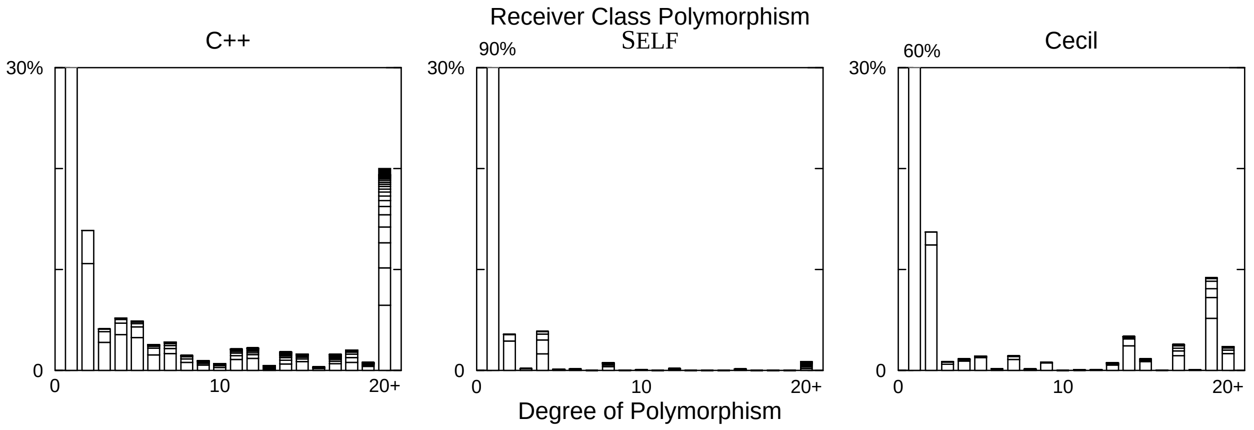


Figure 2: These histograms show the percentage of dynamic sends with a particular degree of polymorphism. Each bar for N receiver classes is divided into N parts, showing the relative frequency from the most common to the least common receiver. The sends with polymorphism greater than 20 are collected into the 20+ bin which accounts for its exaggerated height in the C++ histogram. The monomorphic bar extends to 90% for SELF and to 60% for Cecil.

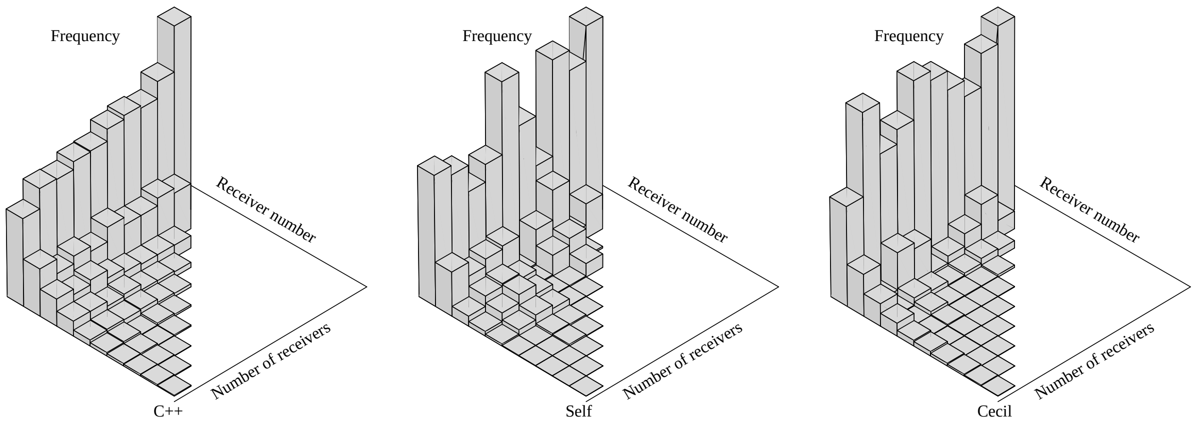


Figure 3: The graphs show the average shapes of the receiver class distributions with a particular degree of polymorphism. The south east-sloping slices represent the average frequency distribution of call sites with a particular degree of polymorphism. Unlike Figure 2, each slice is normalized to 100%.

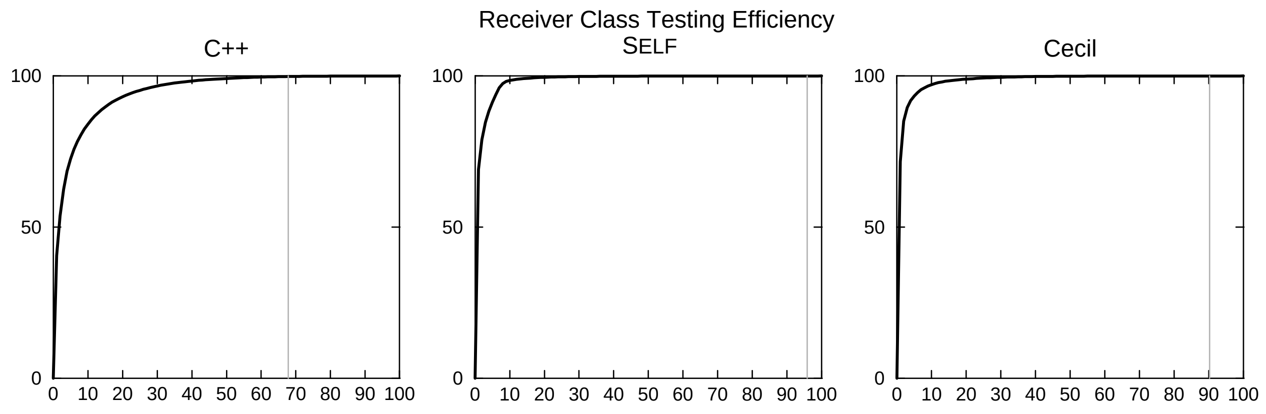


Figure 4: The plot shows the percentage of dynamic sends which would match a class test if a certain percentage of class tests were done, inserting the most effective tests first. The dashed line is drawn where the number of class tests equals the static number of sends in the program text.

wrong. With our measurement methodology, the C++ programs show the greatest amount of polymorphism and SELF programs the least. We believe that the primary reason for the relatively low dynamic degree of polymorphism in these SELF and Cecil programs is due to weaknesses in the implementation technology. With a pure object-oriented language, nearly every operation is a dynamically-bound message, and the compiler must work hard to identify and optimize monomorphic call sites. Despite the SELF compiler's sophisticated techniques, many monomorphic messages remain. In C++, on the other hand, the programmer is constrained to provide a great deal of static information which is then exploited by the compiler to implement most basic operations without dynamic binding. What remains in the C++ profiles are only those operations that the programmer explicitly determined required dynamic binding.

Across all three languages the most common receiver class at a call site receives more than half of the messages sent at that call site. This indicates that programs in all three languages are promising candidates for receiver class prediction. The large number of monomorphic sends in SELF programs illustrates that there is plenty of opportunity for optimizations based on dynamic profile information even in the presence of fancy static analyses.

In both the SELF and Cecil experiments, our decisions of what to measure tend to reduce polymorphism. However, we also performed experiments on SELF without inlining and on Cecil respecting closure identity and the multiple dispatch of fetch and store. The tails of the receiver class distributions extended to a higher degree of polymorphism but the low polymorphism sends were still dominant in both dynamic and static occurrence.

## 3.3 Shapes of Receiver Class Distributions

The three dimensional box plots in Figure 3 more clearly show the difference between the relative frequency of the most common to the least common classes at call sites with a given degree of polymorphism. The data is normalized within the set of call sites with polymorphism *N*, so that each slice sums up to 100% (the information about the relative execution frequency of sites with different polymorphism is suppressed). Each row can be thought of as one of the bars from the histograms in Figure 2 where the vertical slices are spread out along a second axis.

There is a noticeable trend across all three languages for the most common receiver class to dominate all of the less common ones. Although the frequency of the most common class decreases with greater polymorphism, the distribution does not become flat. This indicates that, on average, receiver class prediction can be effective even at highly polymorphic call sites because even one test will catch a large fraction of the sends.

## 3.4 Efficiency of Receiver Class Prediction

Finally, we measure how many dynamic calls can be successfully caught with a certain number of receiver class tests. The cumulative graphs in Figure 4 indicate the fraction of dynamic calls caught for a given amount of tests in the program. The vertical line shows the point which corresponds to an equal number of receiver class tests and call sites (though this does not necessarily correspond to one receiver class test per call site; heavily executed sends may have several receiver class tests while infrequently executed sends may have none). The rate of change of the curve is the marginal benefit per test.

SELF and Cecil have very similar graphs which show that almost all of the sends in the program can be caught with about 10% of the possible receiver class tests. For C++, one would have to insert more tests to catch the same number of sends, but there is still great benefit for the first few tests. These results show that code space increase can be managed by carefully placing tests where they will deliver the most benefit.

### 3.5 Individual Receiver Class Distributions

The data presented in the three previous sections are averaged over several programs and many call sites per program. As such, they tend to obscure the features of individual distributions, which could pose a problem if, for example, the most frequently executed messages had distributions significantly flatter than the average, because these are the messages we are most interested in optimizing. However, this has not happened with any of the programs we have profiled, since we weight messages by their execution frequency when averaging, and the resulting distribution is closer to the distributions of heavily executed messages than to infrequently executed ones. In Figure 5, we present a sample of individual receiver class distributions for the most common messages in a run of our Cecil compiler benchmark. These distributions are at least as promising for type prediction as the average distribution of Cecil runs in Figure 2.

### 3.6 Comparison with Other Studies

**Table 2: Most Common Receiver and Monomorphic Statistics**

| Language | Program | Most Common Receiver | Monomorphic Sends |
|---|---|---|---|
| C++ | `new` SELF compiler | 53% | 13% |
| | `sic` SELF compiler | 57% | 8% |
| | `doc` editor | 86% | 57% |
| | `idraw` graphical editor | 76% | 45% |
| | C++ mean | 68% | 31% |
| SELF | OOSuite benchmarks | 94% | 89% |
| | Cecil compiler in SELF | 98% | 91% |
| Cecil | Cecil compiler in Cecil | 90% | 60% |

Calder and Grunwald also argue that receiver class prediction may be effective for C++ programs. Some of their numbers can be compared directly to ours. For instance, their measure of “Static” sends, or sends which would be correctly predicted by always guessing the most common class, corresponds to the fraction of sends in the bottoms of the bars of the histogram in Figure 2. They report an average of 91% of calls could be so predicted for 7 programs. Our C++ programs had only 68% of their dynamic sends go to the most common receiver class. 66% of the sends in Calder and Grunwald’s programs were at monomorphic call sites, which they call “Single Target” sites, while only 31% of the sends in our C++ programs were monomorphic. These comparisons suggest that our C++ benchmarks we more object-oriented than the programs examined by Calder and Grunwald. Table 2 presents these statistics for all of our benchmark programs.

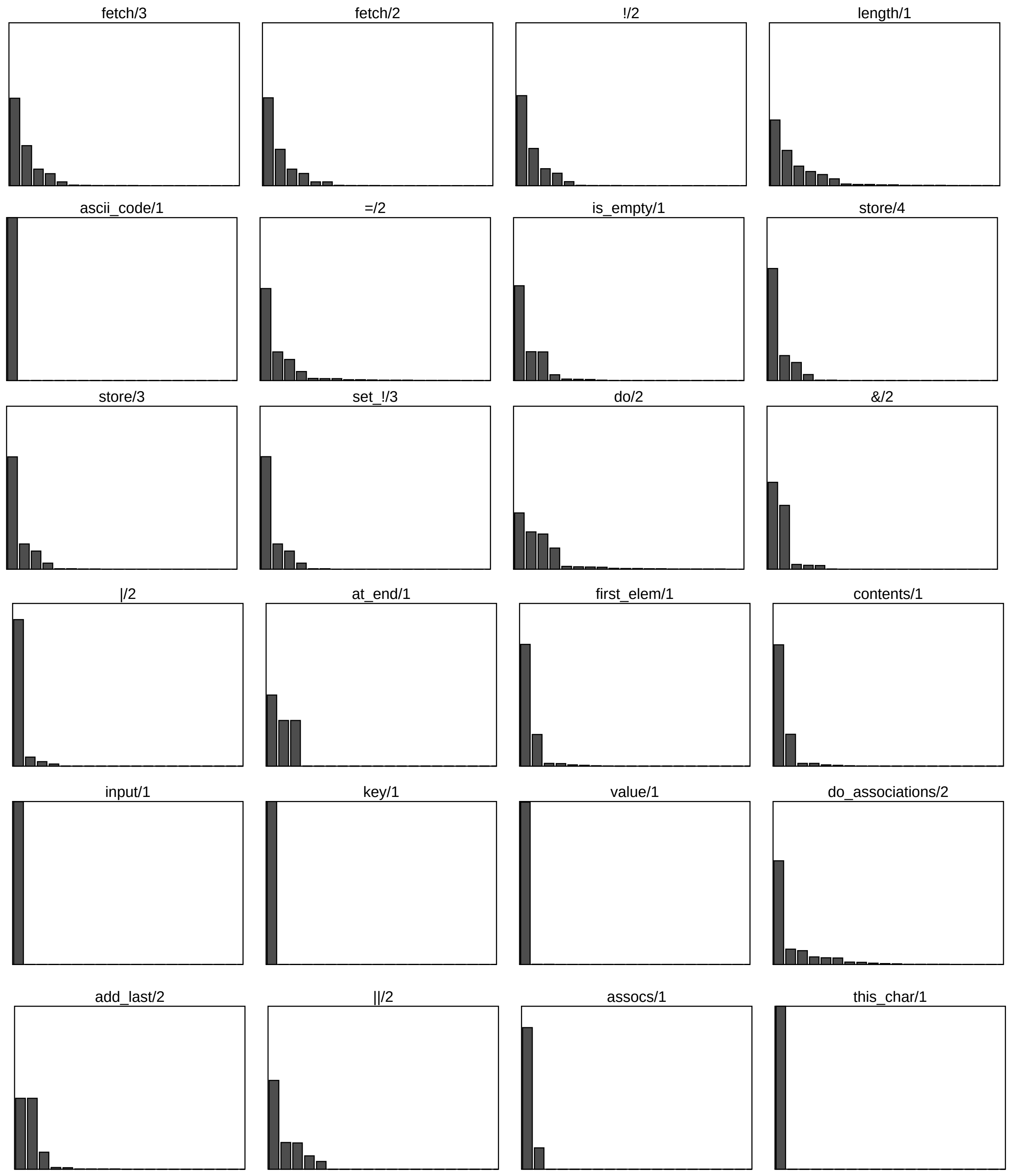


Figure 5: These graphs show the distributions of the 24 most frequently sent messages in a sample run of the Cecil compiler. The labels are in the form message name/number of arguments. Several of the messages were monomorphic and most of them were dominated by the most common receiver class.

## 4 Stability of Distributions

Even if we have a program with very lopsided receiver distributions, profile information is only helpful in optimization if the profile of one run of the program is an effective predictor of the behavior of other runs. We use the term *stability* to refer to the degree to which one program run's profile data predicts the behavior of another program run, and we measure stability by comparing the receiver class distributions of the two runs. We studied three levels of stability: stability of profiles of the same program for different inputs, stability of profiles of different versions in time of a program, and stability of library code across different programs. We expect that profiles will be most stable for different runs of the same program but less stable from version to version of a program and across programs sharing a library.

We are interested in two kinds of stability. First, there is the stability of receiver class distributions: did the same classes occur at a call site, and were they in the same order of relative frequency. We desire stable receiver class distributions so that the receiver tests inserted as a result of one profile are effective when executed in other runs. Second, there is the stability of the execution frequency of the call site or the message. If a call site was heavily executed in one run but not in another, then we might waste time by optimizing it for the second run.

### 4.1 Methodology

We measure stability of receiver class distributions in several ways. One metric is the $L_2$ difference[*] between two normalized distributions, which is a very good indicator of stability when the difference is close to its maximum or minimum, but is not as useful when it is in the middle. This metric has the advantage that it is independent of any application of the receiver class distributions. The other two metrics are geared more towards our intended application of the data to guide insertion of class tests. The OrderSame metric classifies two distributions as the same if they are comprised of the same receiver classes and in the same order of decreasing frequency; two distributions considered the same according to OrderSame would likely lead to the same sequence of inserted class tests. Since the first receiver class is the most important, we include a third metric, FirstSame, which classifies two distributions the same as long as their most common receiver classes are the same.

We measure the stability of execution frequency by ordering all of a program's call sites and computing the correlation between the positions of call sites in the ordering from one program to another. If the call sites are similarly ordered, then they will lie close to a straight line.

We were able to measure the stability of Cecil and C++ programs. For cross-input experiments, we used the Cecil compiler and the two InterViews editing programs, as listed in table 1. We could only measure cross-version stability on the Cecil compiler, because we have only a single version of the other programs. We were able to measure cross-program stability in Cecil programs as well. We discuss the impact of our choice of benchmarks on the stability results in each subsection.

[*] If distributions with $n$ receiver classes are considered as points in an $n$-dimensional space, the $L_2$ norm is the Euclidean distance between two points.

## 4.2 Stability Across Inputs

We measured the Cecil compiler compiling two different programs in two different compiler configurations (with and without inlining). We report in this section comparisons between two of these four profiles; the other 5 pairwise comparisons are nearly the same.

We believed that the receiver class distributions observed for these four profiles will be similar, and the results bear out this expectation. Considering call site specific distributions, over 99% of the call sites executed for one input were also executed for the other. 80% of the call sites had the same distributions according to the OrderSame metric while over 99% were the same according to the FirstSame metric. For global message distributions, again more than 99% of the messages were in common between the two inputs, 64% were the same by the OrderSame metric, and over 99% were the same by the FirstSame metric.

The two histograms in Figure 6 (a,b) show that the profiles were also very similar according to the $L_2$ norm. The call site specific distributions barely changed at all, and the global message distributions were also extremely stable. Messages change more than call sites because it only takes a change at one call site to change a message, such as a new receiver class appearing at one call site.

The scatter plots of message and call site frequency in Figure 6 (c, d) show that the relative frequency of calls is similar for both inputs, with most of the change in frequency happening at the low frequency end. That means that it is likely that we would want to optimize the same call sites and messages for both inputs. In terms of frequency, messages are somewhat more stable than call sites, because the relative execution frequency of a message is averaged across many call sites.

Putting both receiver class distribution and execution frequency results together, the Cecil compiler should perform well when optimized using one profile and compiling other input programs.

We also measured the InterViews programs, `doc` and `idraw` running on two different inputs each. We attempted to make the inputs fairly distinct for each program. For the `doc` program, one input was to enter a page of text from scratch and save it, and the other input was to load a ten page document with figures and cut and paste selections from one place to another, also saving the result. The `idraw` program provides the usual assortment of graphical primitives, rectangle, lines, ellipse, text, etc. and the usual transformation operations, scale, move, rotate, edit point, and so on. One of our inputs was to create only rectangles and apply a variety of operations to them. The second input consisted of creating one object of each primitive type and applying each kind of operation to it. So for example, if the scale operation sends messages to the object being scaled, then in the one case, those call sites should be monomorphic with rectangle as the sole receiver class, and in the other case, the call sites should be polymorphic and approximately evenly distributed over all receiver classes.

Our results, averaged over both programs, are shown in Figure 7, where it is apparent that the C++ programs are fairly stable across inputs, but not as stable as our Cecil programs. 99% of the call sites executed for one input were also executed for the other. 45% of the call sites were the same according to the OrderSame metric, while 79% were identical by the FirstSame metric. For global message distributions, we observed that 99% of the messages occurring for one input were

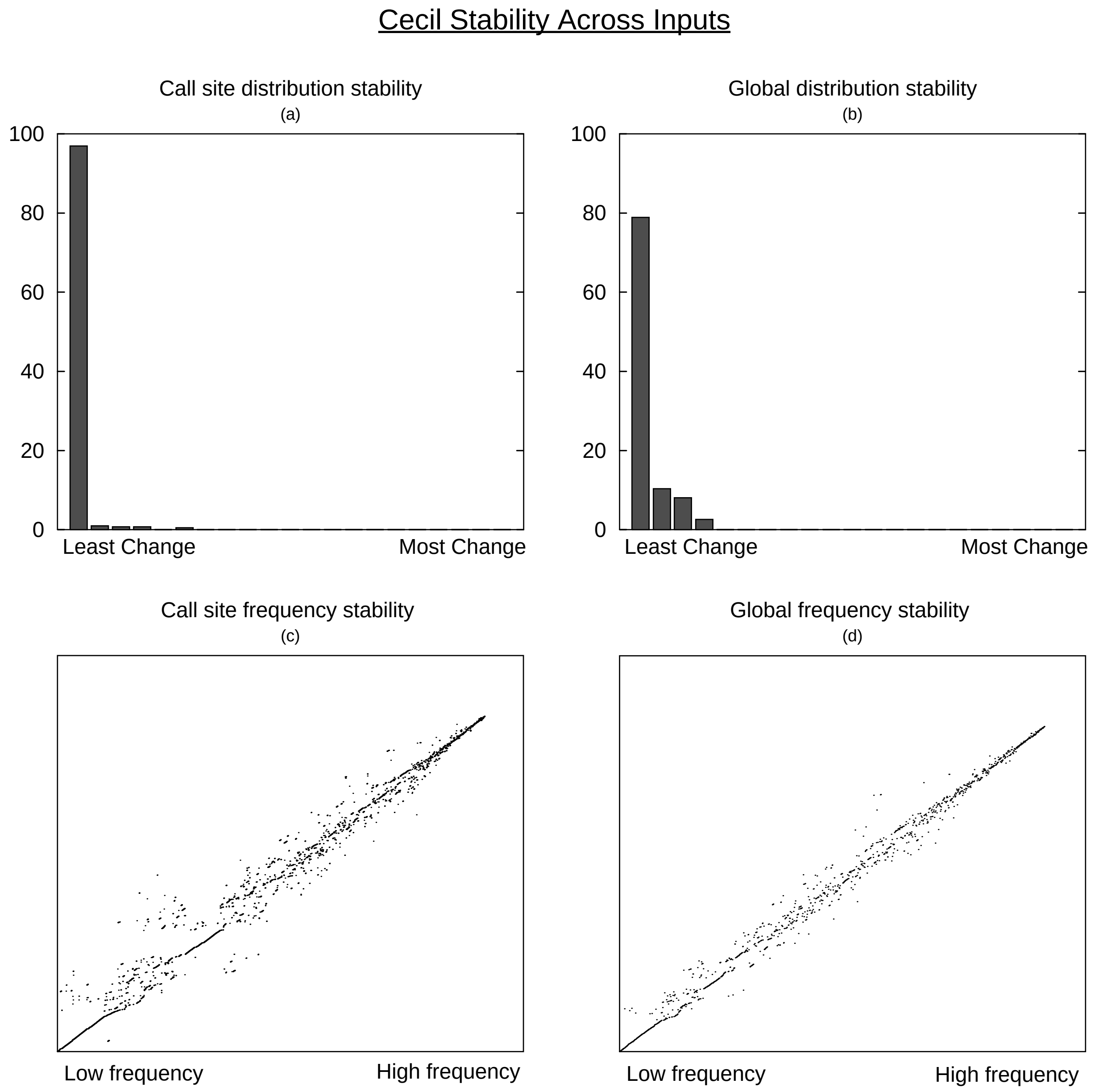


Figure 6: The histograms (a) and (b) show the percentage of dynamic call sites and messages that change by some amount across inputs. The change is measured using the $L_2$ norm, so least change is an extremely stable distribution and most change is a distribution which had no classes in common for the two input files. The scatter plots (c) and (d) show the change in call site (message) execution across inputs. The x-axis is the call site's (message's) position in a sorted order of all call sites (messages) for one input and the y-axis is its position for the other input. The more closely the points lie to a straight line, the more stable the execution frequency is.

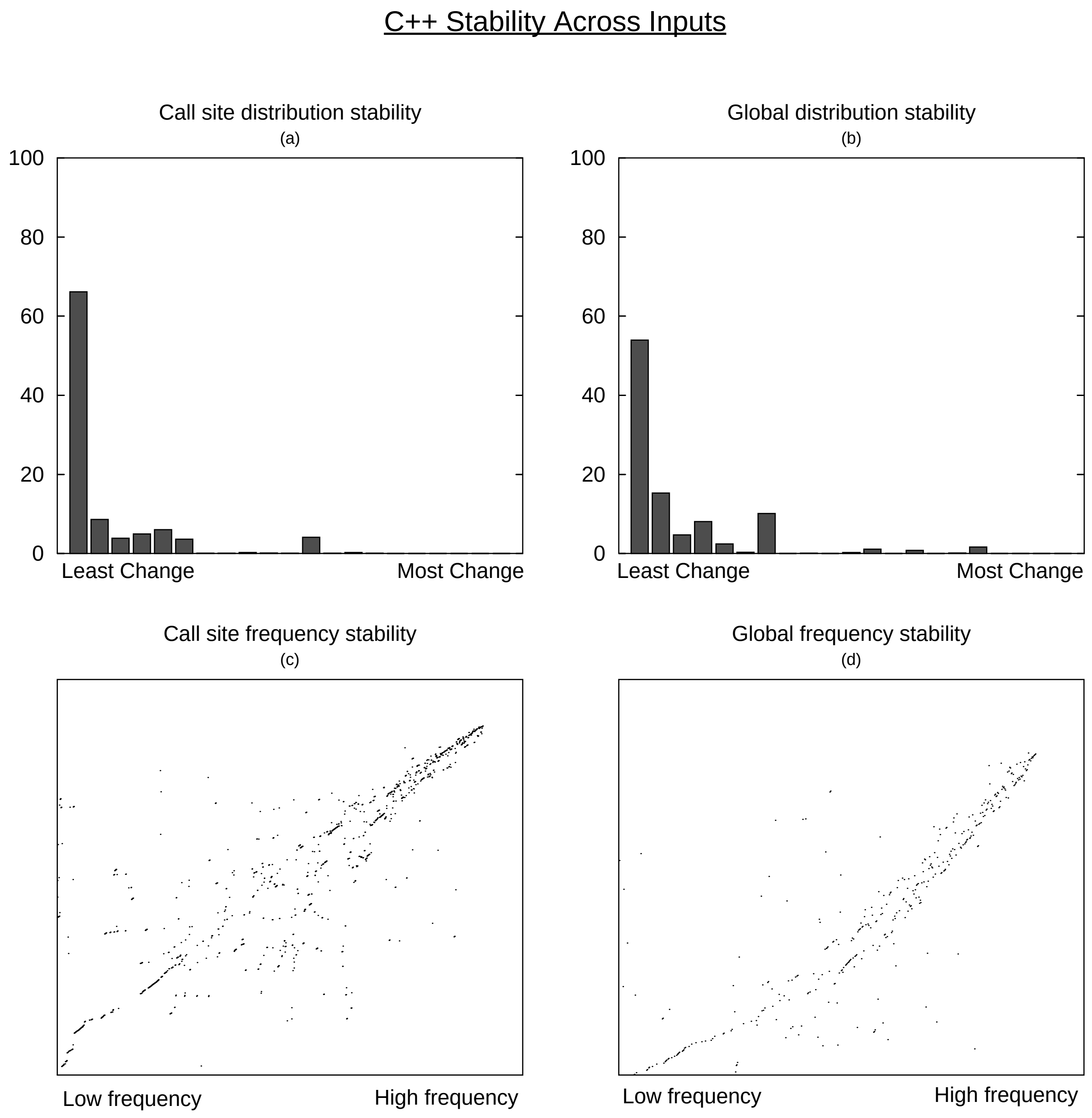


Figure 7: The histograms (a) and (b) show the percentage of dynamic call sites and messages that change by some amount across inputs for our InterViews programs, `doc` and `idraw`. The change is measured using the $L_2$ norm, so least change is an extremely stable distribution and most change is a distribution which had no classes in common for the two input files. The scatter plots (c) and (d) show the change in call site (message) execution across inputs. The x-axis is the call site's (message's) position in a sorted order of all call sites (messages) for one input and the y-axis is its position for the other input. The more closely the points lie to a straight line, the more stable the execution frequency is.

repeated for the other input. Also 28% of the messages had identical distributions under the OrderSame metric and 99% were the same according to the FirstSame metric. The histograms of figure 7 (a) and (b) also show that the programs were stable across inputs. The frequency distributions in parts (c) and (d) are not as stable as their Cecil counterparts, but the call sites and messages did tend to maintain their relative frequency.

We expected that the Cecil compiler would be very stable across inputs, because compilers must perform many of the same actions for all inputs. The document preparation and graphical editor programs were chosen specifically because we believed that they would exhibit more variation across inputs. In fact, they did vary more, but most call sites and messages would still be correctly predicted from either profile. Therefore, we believe that receiver class prediction should work well for a variety of programs, whether they are batch-oriented like a compiler or interactive like a graphical editor.

## 4.3 Stability Across Program Versions

For profile-guided optimization to be most useful in a program development environment, it would be desirable if profile information gathered for one version of a program could be reused to optimize future versions of the program as it evolves. To assess how stable receiver class distributions were from one version of a program to another, we compared four snapshots of the Cecil compiler over a one month period of rapid development. Table 3 describes each compiler version and the changes made between different versions.

**Table 3: Cecil Compiler Versions Profiled**

| Date | Size (lines) | % Lines Changed From Baseline | Changes from previous version |
|---|---|---|---|
| December 11 | 17,110 | n/a | Baseline version. Includes fully functional parser and code generator. |
| December 20 | 18,452 | 13% | Small changes to format of generated code. Simpler environment creation code. Most uses of == changed to =. File_streams changed to write data more frequently. Parser bug fixed. Lower case coercion added for characters and used in scanner. |
| January 3 | 19,645 | 22% | Set hierarchy reorganized, list_sets and hash_sets introduced and used throughout the compiler. Evaluation of expressions on command line. Debugging extended with access to runtime environments. New array and vector creation methods introduced and used widely. |
| January 12 | 20,215 | 28% | New closure id's and new field in PICs altered generated code format slightly. Method table data structure changed with new method_identifier class. |

We expected that there would be greater change across versions than across inputs, because new classes and methods were introduced into the compiler which would not be predicted by old profiles. The cross-version histograms of Figure 8 (a,b) show the change in the Cecil compiler's profile from the first version to each of the 3 later versions in terms of the $L_2$ metric. As expected, the profiles showed decreasing stability over time as more changes accumulated. But even the difference between the first and last versions was not very large. There are some call sites which change as much as possible by this metric, meaning that every receiver class in the second version was different from the first version and all class tests based on the first profile would miss for the second program. This results from new classes introduced by program revisions which could not be predicted by any earlier program. Once again, the individual call sites are slightly more stable than messages, for the same reason as above.

The frequency stability graphs, Figure 8 (c,d), show that across versions, the high frequency call sites and messages tend to remain high frequency. There are some individual points where call sites and messages changed frequency significantly, but there is no large-scale trend for call sites to change.

The graphs in Figure 9 depict the change over time in terms of the OrderSame and FirstSame metrics. The OrderSame metric is quite restrictive, and consequently a large fraction of call site specific and global message distributions were considered to have changed. Under the OrderSame metric, the changes from one version to another do not sum, however; profiles age rapidly at first but then do not degrade much after that. We attribute this behavior to the expectation that the parts of the compiler that are modified during one week are likely to continue to be modified in later weeks, and conversely code that is unchanged during one week is likely to remain unchanged in later weeks.

According to the FirstSame metric, very few distributions changed from one version to another, particularly when considering global message distributions. This indicates that optimizations such as receiver class prediction that depend only on the most commonly-occurring classes may still be effective when using profiles from older versions of a program.

## 4.4 Stability Across Programs

We expect that in the future shared libraries and frameworks [Wirfs-Brock & Johnson 90, Deutsch 89] more and more will form common building blocks from which applications are composed. We wish to learn whether a profile of a library included in one program is a good predictor of the behavior of that library when included in another program. If so, then we can store receiver class distribution information along with libraries so they can be optimized whenever they are included into an application.

To assess the stability of Cecil profiles across programs, we divided up the Cecil compiler into two pieces, the parser and the code generator, and then profiled how these two “programs” used the Cecil standard data structure library. We also compared the two C++ programs, `doc` and `idraw`, which share the InterViews library. We expected this kind of stability to be the lowest of the three kinds we measured, but hoped that some stability would remain.

For the two Cecil programs, our measurements show that receiver class distributions are still remarkably stable across programs, but that execution frequency is not at all stable. The two programs had about 85% of their call sites and 95% of their messages in common. Furthermore, 65% of the call sites and 50% of the messages have unchanged distributions according to the

## Cecil Stability Across Versions

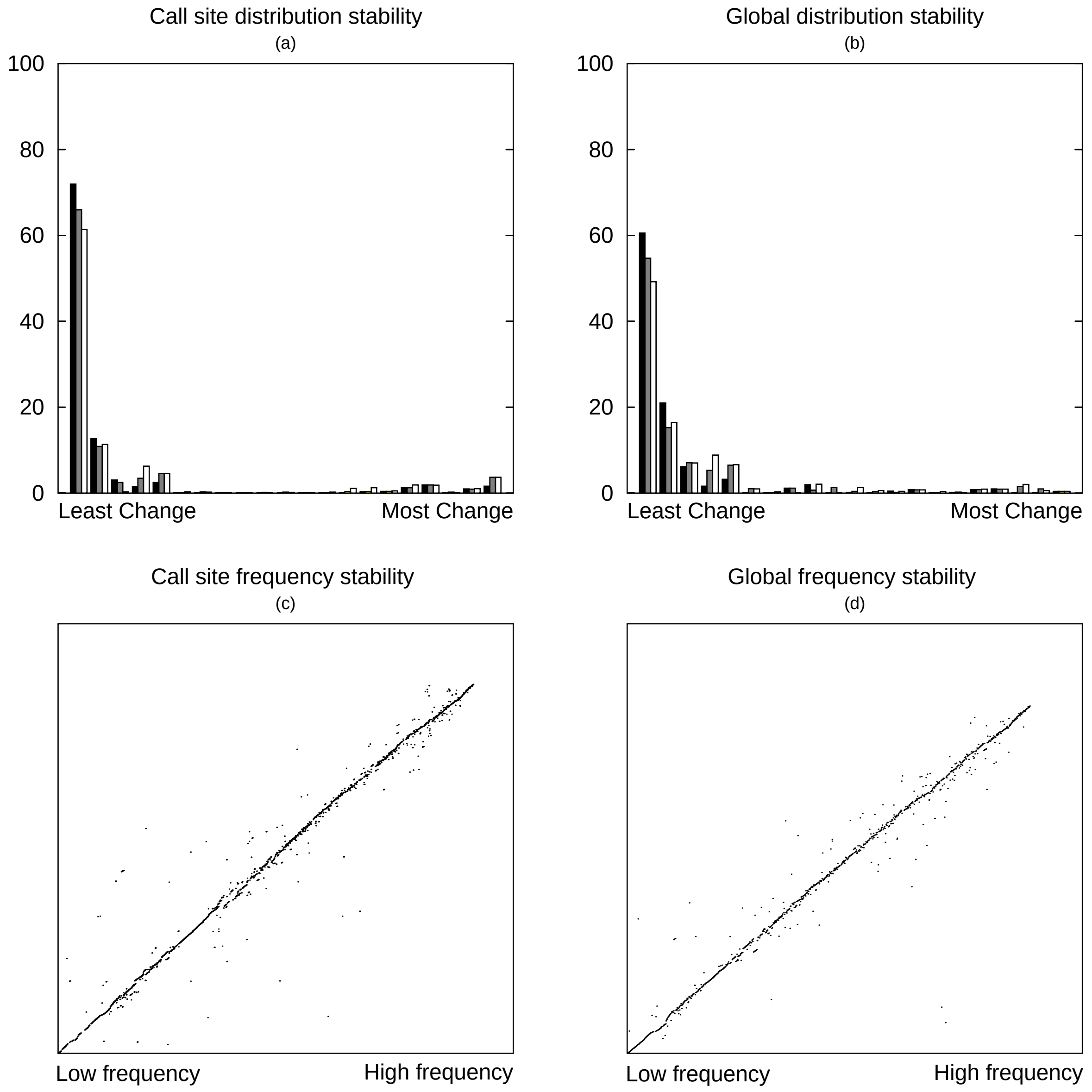


Figure 8: The histograms (a) and (b) show the percentage of dynamic call sites and messages that change by some amount across versions. Since we have 4 versions of the Cecil programs to profile, we have drawn 3 histograms together, showing the change between the first version and the second, third and fourth versions. The change is again measured using the $L_2$ norm. The scatter plots (c) and (d) show the change in call site (message) execution across versions. They are comparisons of the first and fourth versions, so they should have the greatest change.

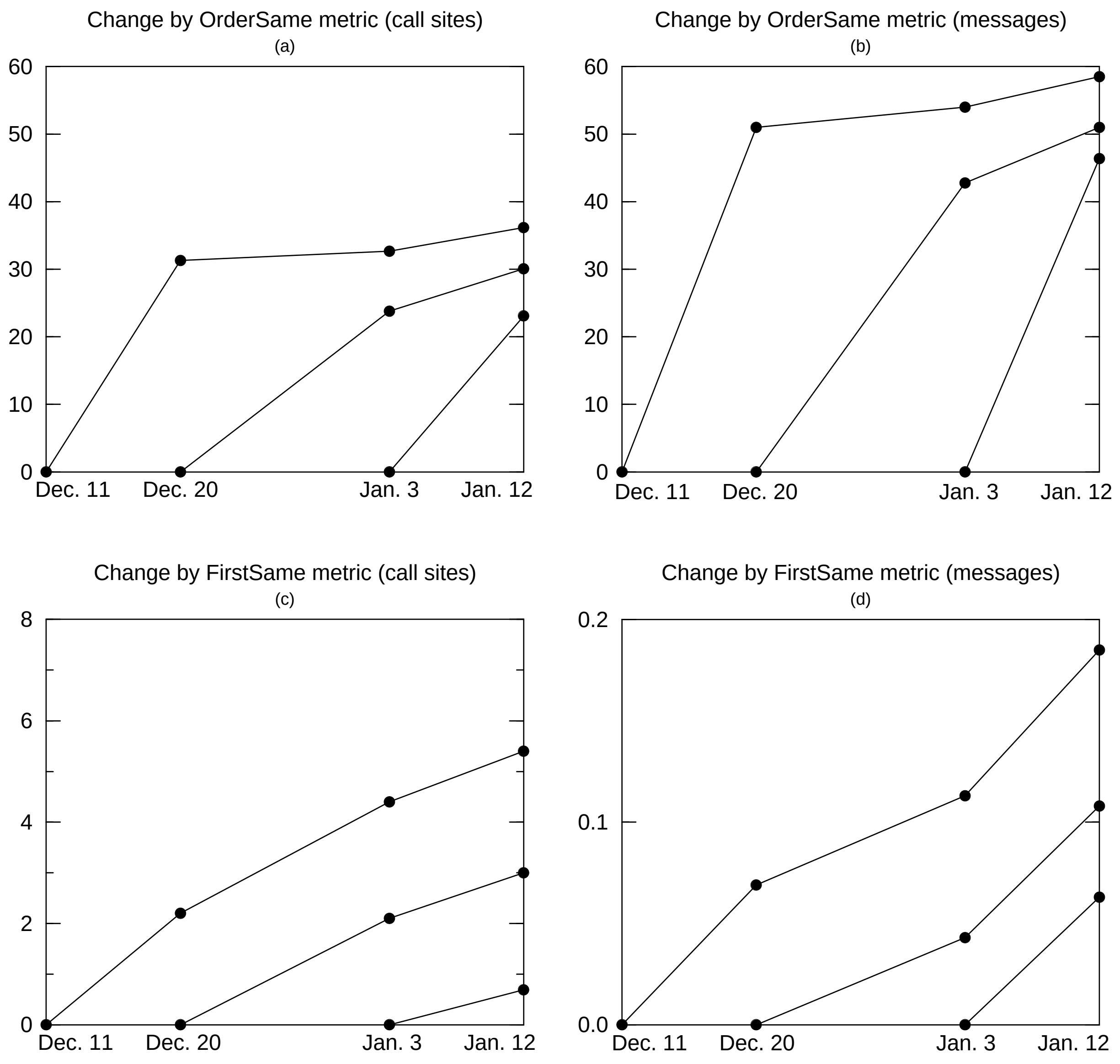


Figure 9: These plots reveal the change in call sites and messages across versions using two different metrics of change. Graphs (a) and (b) show the change according to the OrderSame metric at call sites and messages, respectively. The changes in call sites and messages according to the FirstSame metric are shown in (c) and (d). In both rows, the x-axis is the date of the compiler version and the y-axis is the percentage of changed call sites. Each dot represents one pairwise comparison between two versions. The lines show how changes accumulated as more changes were made to the programs.

OrderSame metric (comparable to the cross-version results), and 83% of the call sites and 94% of the messages have unchanged distributions according to the FirstSame metric. Therefore we would expect to be able to optimize many of the call sites in the standard library effectively based on the profile of one program (of course, collecting profiles from other programs in addition will only improve these results).

The histograms in Figure 10 (a,b) show the changes in distributions according to the $L_2$ norm. Although some call sites and messages did not change across programs, the ones which did change tended to change a lot. Further bad news is found in the frequency stability pictures, Figure 10 (c,d) which show that frequencies changed a great deal between programs, so that even though some class distributions could be accurately predicted, the relative frequency of those call sites could not. This implies that call sites in the standard library can be optimized using the receiver class distribution information, but that the compiler should not assume that an infrequently executed call site will remain so in other programs and so be undeserving of optimization effort.

For the two InterViews programs, we found much less similarity between the receiver class distributions or execution frequencies. Only 24% of the call sites and 55% of the messages occur in both programs, so there is only a small possible benefit from class prediction. Furthermore, just 14% of the call sites and 23% of the messages have identical distributions according to the OrderSame metric. By the FirstSame metric, 23% of the call sites and 54% of the messages have matching distributions. The graphs of Figure 11 (a, b) tell the story in more detail. In graph (a), all bars of the histogram are small because of the fact that only 24% of the call sites were shared between the programs, so all of the bar heights add up to 0.24. The frequency stability graphs (c) and (d) indicate that those call sites and messages which were shared did not occur with similar frequencies in both programs. These measurements show that the cross-program stability we found for two Cecil programs does not hold true in general, so we cannot rely on receiver class prediction to optimize a shared library well for all programs.

### 4.5 Comparing Global to Call Site Specific Receiver Class Prediction

Call site specific distributions are more precise when they apply than global message distributions. Fortunately, we can use the same distribution comparison techniques to assess how good a predictor a message's global receiver class distribution is of the call site specific distributions of the call sites that send that message.

The histogram in Figure 12 shows that the global message distribution for the Cecil compiler written in Cecil is an excellent predictor of the call site specific distribution for about 65% of the calls, dynamically; the rest of the calls scattered between pretty good and very bad. Call site specific information, when available, is almost always a better predictor of a call site's future behavior than is global message information, but in its absence global message distributions appear to be an able substitute.

## 5 Impact of Receiver Class Distributions on Performance

To assess the performance improvements derived from applying dynamic profile data to guide receiver class prediction, we measured the run-time performance of several Cecil programs compiled with the following levels of optimization:

## Cecil Stability Across Programs

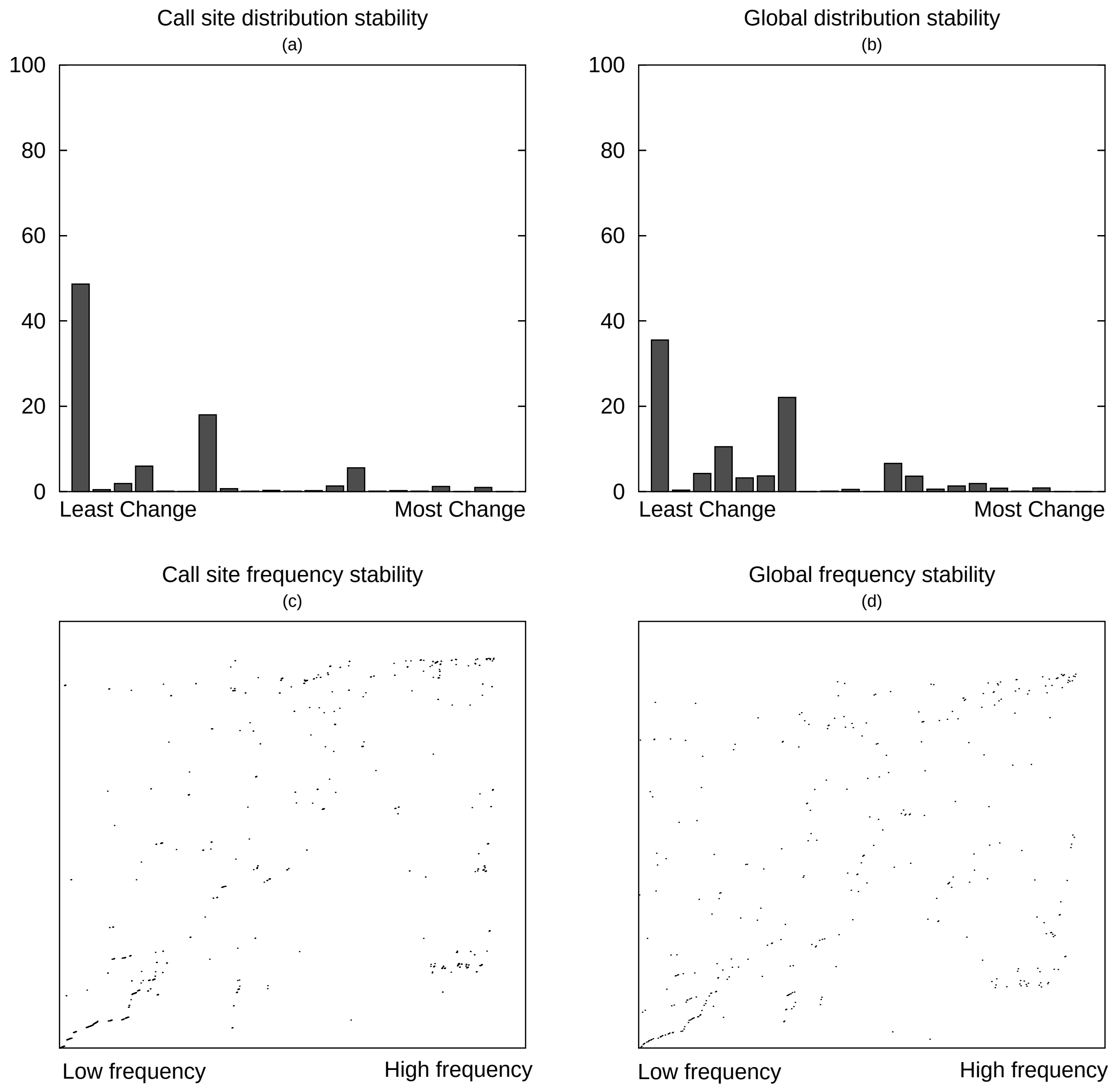


Figure 10: The histograms (a) and (b) show the percentage of dynamic call sites and messages that change by some amount across programs. The scatter plots (c) and (d) show the change in call site (message) execution across programs. In contrast to the cross input and cross version plots, the cross program plots are not close to a straight line so the execution frequency from one program does not predict the frequency in the second program.

## C++ Stability Across Programs

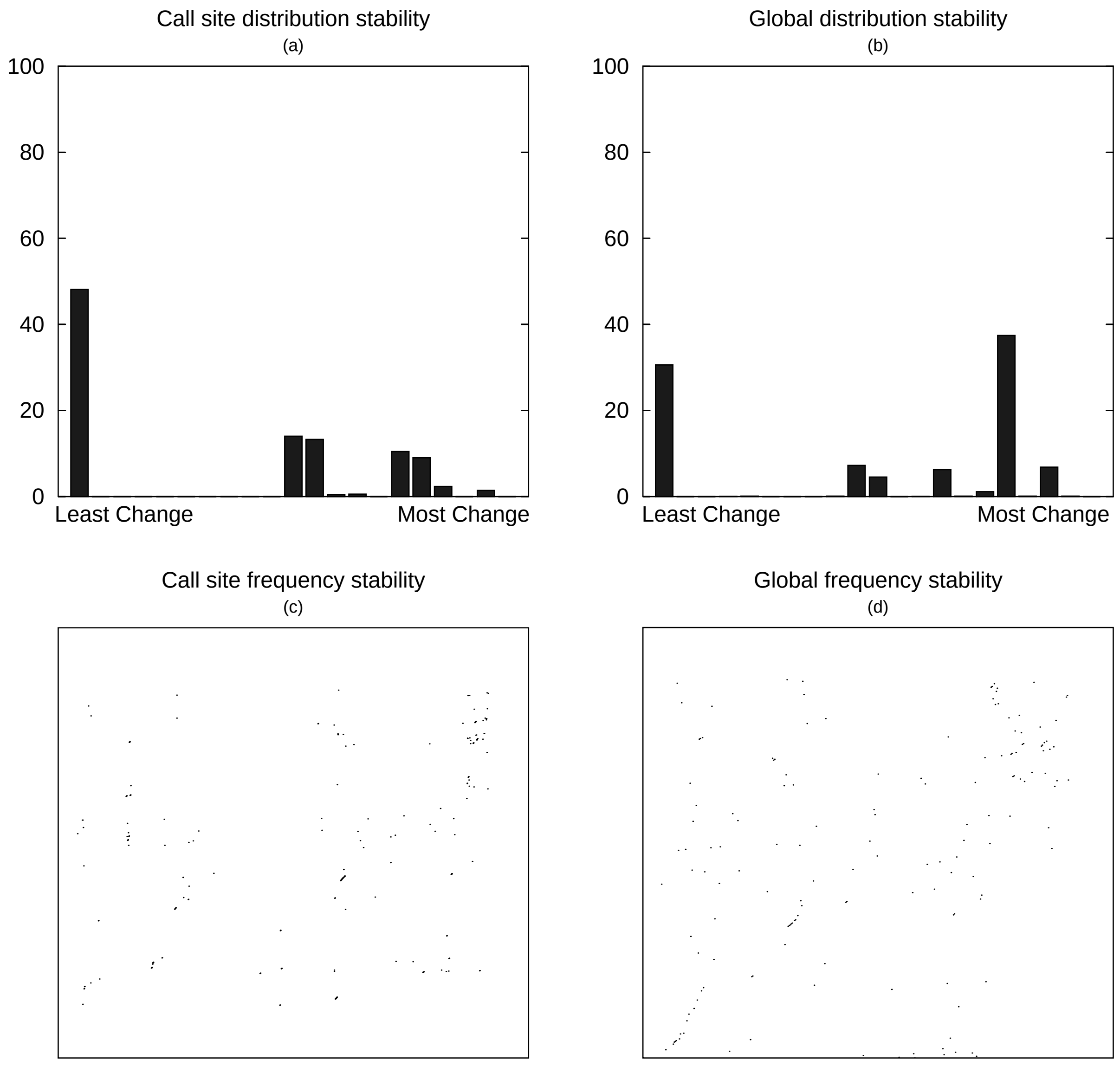


Figure 11: The histograms (a) and (b) show the percentage of dynamic call sites and messages that change by some amount across programs. The scatter plots (c) and (d) show the change in call site (message) execution across programs. In contrast to the cross input and cross version plots, the cross program plots are not close to a straight line so the execution frequency from one program does not predict the frequency in the second program.

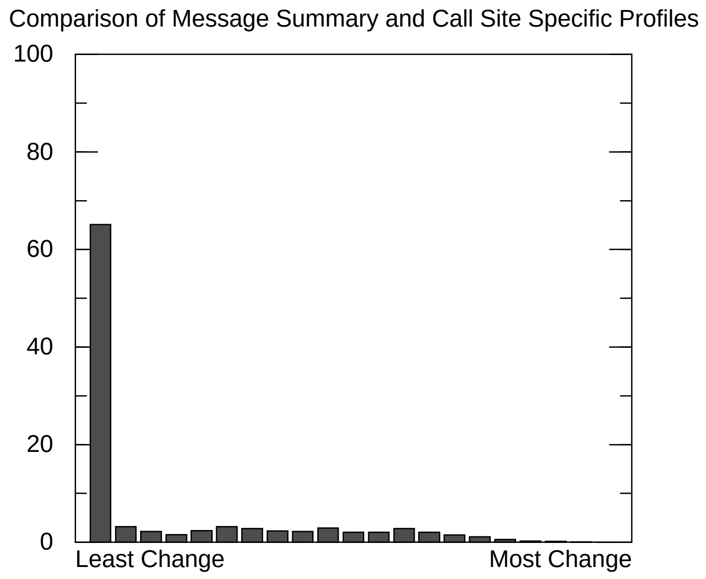


Figure 12:This graph is similar to the preceding stability histograms, but it shows the similarity between the receiver class distributions for messages and distributions for call sites in the same program run. A small change means that the message distribution is a good predictor and a large change means that it would predict the wrong receiver classes for a call site.

- no optimization;
- full static optimization (including static concrete type analysis, dead assignment elimination, closure creation delaying, and automatic inlining) plus receiver class prediction guided by a small, fixed global message distribution supporting receiver class prediction for messages such as `+` and `if` (this configuration simulates a Cecil implementation incorporating Smalltalk-80 hard-wiring or SELF-like type prediction techniques);
- full static optimization plus receiver class prediction using a global message distribution for the benchmark; and
- full static optimization plus receiver class prediction using a call site specific distribution for the benchmark.

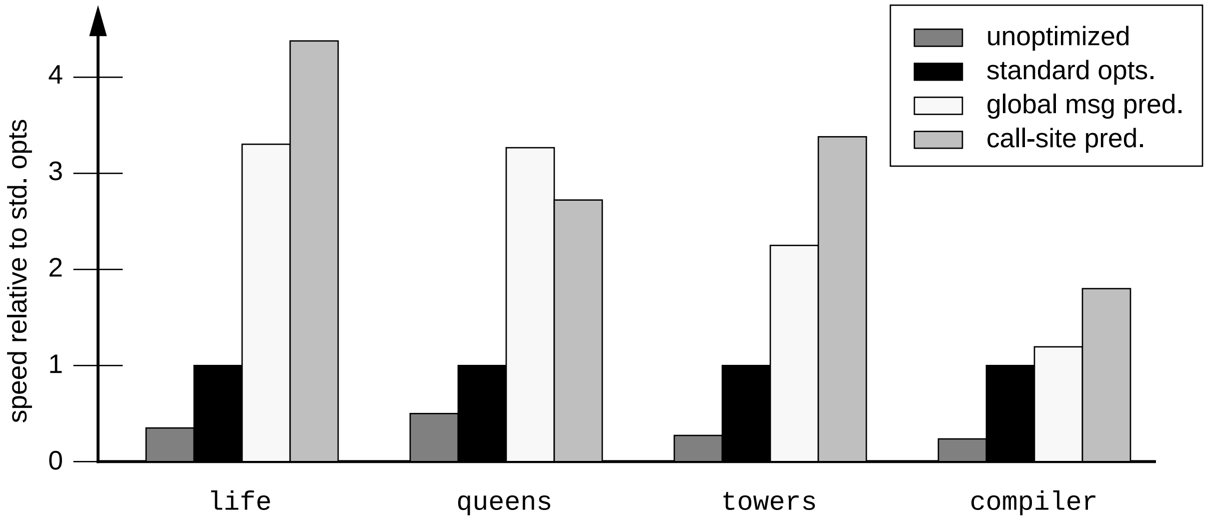
The current Cecil compiler does not include customization or splitting [Chambers *et al.* 89], two techniques included in the SELF compiler that would improve absolute performance. Because customization transforms polymorphic sends into monomorphic sends statically, the presence of customization would be likely to reduce somewhat the need for profile-based optimizations. Splitting, on the other hand, might increase the effectiveness of receiver class prediction by eliminating redundant class tests and thus gaining the same benefit with fewer tests. Unlike the SELF compiler, however, the Cecil compiler exploits full knowledge of the class inheritance graph during compilation. In particular, when compiling a method associated with a particular class, the compiler knows statically that instances of only the class and its subclasses can be associated with the receiver formal parameter. Often this provides much of the same information as does customization in SELF. Future work includes exploring the impact of these and other static optimizations on the effectiveness of profile-guided optimization.

The following table presents the performance results of applying profile-based class prediction to several Cecil benchmark programs:

**Table 4: Execution Time of Cecil Benchmarks with Varying Optimization Levels**

| **Benchmark** | **Size** | **Compiler Configuration** | | | |
|---|---|---|---|---|---|
| | | **Unoptimized** | **Standard prediction** | **Global message prediction** | **Call-site specific prediction** |
| `towers` | 92 lines + 6,000-line library | 2,110 ms | 830 ms | 250 ms | 190 ms |
| `queens` | 162 lines + library | 5,780 ms | 2,830 ms | 870 ms | 1,040 ms |
| `life` | 166 lines + library | 21,180 ms | 5,860 ms | 2,590 ms | 1,720 ms |
| `compiler` | 23,400 lines + library | 5,514 s | 2,332 s | 1,979 s | 1,304 s |

The following chart shows the relative improvement with different levels of profile input:



The results indicate that profile-based class prediction can yield significant performance improvements. As expected, programs optimized with call-site specific prediction typically outperform programs optimized with global message prediction, which in turn outperform programs optimized with only standard prediction.

For the `queens` benchmark, the call-site specific prediction version actually ran slower than the global message prediction version. We believe this is an artifact of the direct-mapped instruction cache on the SparcStation IPX used for benchmarking. Profiling the two programs with QPT [Ball & Larus 92] revealed that the call-site specific prediction version of `queens` executed substantially fewer instructions than the global message prediction version.

## 6 Future Work

Several issues arise when considering profiling optimized programs. One issue is how to gather profile information from an optimized program. After receiver class prediction has been applied, many messages are no longer handled as dynamic sends, so dumping out the PIC runtime dispatching structures will miss many of these common messages. Clearly instrumentation code could be added to also record successfully predicted messages, but the performance impact of the additional monitoring code could be significant. Another potential problem is that, after optimizations such as inlining, there can be many inlined copies of a particular source method, each with its own sends and corresponding profile data. To reconstruct an unoptimized distribution, the distributions of each of the inlined copies of a send could be added together to form a single unoptimized distribution.

A different issue is that profiles of unoptimized programs can be less precise than information computed statically. For example, in an unoptimized program, there is only one version of any given routine *R*, and the profile of a message within *R* will "smear" together the information derived from the callers of *R*. If *R* is a polymorphic routine, then each caller of *R* is likely to contribute very different information to the profiles of messages within *R*. In an optimized program, inlining, customization, and other optimizations can lead to multiple versions of *R* being compiled. When applying receiver class prediction to the messages within some version of *R*, the compiler would be ill advised to use the "smeared" profile of the unoptimized version of *R*, since these profiles have been diluted with other calls to *R*. If profiles are taken from optimized programs, then perhaps the corresponding optimized version of *R* in the profiled version could be used to derive the distributions for the messages in *R*.

Interactions between profile-guided optimizations and other static optimizations such as interprocedural static class analysis, customization, and splitting merit further study. These studies could examine both the different strengths and weaknesses of static vs. dynamic sources of information, as well as investigating how dynamic information can be applied to guide the application of static effort.

Receiver class profile information can be used in other ways besides optimizing a program. Like other kinds of profile data, it provides feedback which the programmer can compare against his or her expectations. For example, the `do` method for lists in Cecil is implemented recursively, so that it calls itself on the tail of the list until the tail becomes the `nil` object. We found that in one profile, this call site sent the `do` message to `list` objects 60% of the time and to `nil` objects 40% of the time, which indicates that the average length of the lists we iterated over was less than 2 items. This behavior was correct for our program, although we had not predicted it in advance. In other situations, receiver class profiles could identify incorrect behavior.

## 7 Conclusions

Our experiments indicate that programs in C++, SELF, and Cecil are promising candidates for receiver class prediction. Static analysis as performed in SELF and Cecil does not seem to reduce the opportunities for class prediction. Since program execution time is usually spent in a small fraction of call sites, we can achieve most of the benefits of class prediction by optimizing at the most frequently executed sites and we can also restrict our tests to the single most common class at most sites. Furthermore, we have found that the stability of C++ and Cecil programs across different inputs and our Cecil benchmark across versions is sufficient to accurately predict the most frequently executed call sites and the receiver class distributions occurring there. Measurements of the performance improvement of a large, heavily-used Cecil program shows that real object-oriented programs can be sped up by a factor of two with profile-guided receiver class prediction. Profiles of object-oriented programs have other applications in compilers for object-oriented languages, such as helping to determine where optimizations such as inlining or customization are most profitable [Dean *et al.* 94].

## 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. We would like to thank Stephen North and Eleftherios Koutsofios of AT&T Bell Laboratories for producing `dot`, a program for automatic graph layout. `dot` has been helpful in debugging large control flow graphs.

## References

[Ball & Larus 92] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, January, 1992.

[Calder & Grunwald 94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead In C++ Programs. In *ACM Principles and Practice of Programming Languages*, Portland, OR, January, 1994.

[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 *OOPSLA '89 Conference Proceedings*, pp. 49-70, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

[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 92] 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.

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

[Chang *et al.* 91] Pohua P. Chang, Scott A. Mahlke and Wen-Mei W. Hwu. Using Profile Information to Assist Classic Code Optimizations. In *Software-Practice and Experience 21(12)*, pp. 1301-1321, December, 1991.

[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.

[Dean *et al.* 94] Jeffrey Dean, Craig Chambers, and David Grove. Identifying Profitable Specialization in Object-Oriented Languages. To appear in *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, FL, June, 1994.

[Deutsch & Schiffman 84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 297-302, Salt Lake City, UT, January, 1984.

[Deutsch 89] L. Peter Deutsch. Design Reuse and Frameworks for the Smalltalk-80 System. In *Software Reusability*, Vol. II, T.J. Biggerstaff and Alan. J. Perlis, eds., ACM Press, pp. 57-71, 1989.

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

[Graham *et al.* 82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 120-126, Boston, MA, June, 1982. Published as *SIGPLAN Notices 17(6)*, June, 1982.

[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.

[Hölzle & Ungar 94] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. To appear in *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June, 1994.

[Rose 88] John R. Rose. Fast Dispatch Mechanisms for Stock Hardware. In *OOPSLA '88 Conference Proceedings*, pp. 27-35, San Diego, CA, October, 1988. Published as *SIGPLAN Notices 23(11)*, November, 1988.

[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.

[Wall 91] David W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. In *Proceedings of the SIGPLAN ‘91 Conference on Programming Language Design and Implementation*, pp. 59-70, Toronto, Canada, June 1991. Published as *SIGPLAN Notices 26(6)*, June 1991.

[Wirfs-Brock & Johnson 90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying Current Research in Object-Oriented Design. In *Communications of the ACM 3(9)*, pp. 104-124, September, 1990.

## Appendix A Profiling C++ Programs

In order to profile SELF and Cecil programs, we took advantage of built-in PIC data structures, but C++ has no similar built-in facility to record receiver classes. Not wanting to alter the C++ compiler itself to enable profiling, we devised a way of instrumenting the generated code to record receiver classes. We used the `g++` compiler, but similar techniques should work for other compilers.

Each C++ class with virtual member functions has a virtual function table, depicted below, which is used in dynamic binding, and each object of such a class has a pointer to the table. When a virtual function is called, this base pointer is loaded, then an offset depending on the function name is added to the base pointer and the resulting element of the table, the member function address, is loaded. Finally a call is made to that address. To profile a virtual function call, we insert a static call just before it which records the value of `base + offset` and the return address. When the program exits, we store the profile to disk. If we need to determine the symbolic names of the call sites and receiver classes, we can use symbol table information. The symbolic name of a virtual function table contains the receiver class name and the symbolic name of a member function contains the message name.

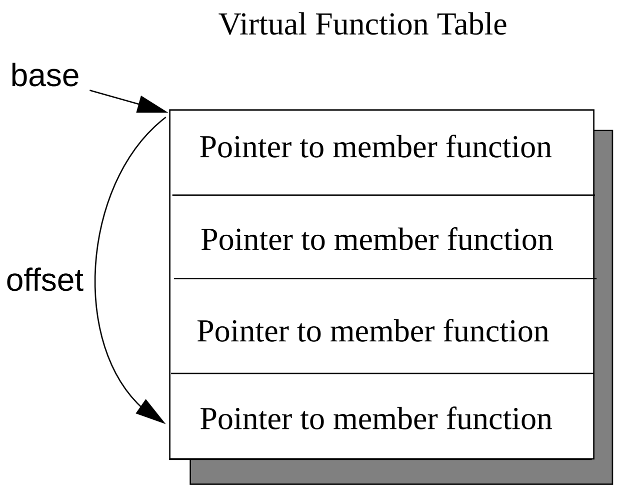

Virtual Function Table
base
offset
Pointer to member function
Pointer to member function
Pointer to member function
Pointer to member function

Figure A1: Structure of C++ virtual function table. Given the value of base + offset, we can determine the receiver class, message name and invoked method from the program's symbol table.

The `g++` compiler generates the following sequence of assembly language statements which enables us to identify virtual function call sites:

```
load  [register + offset], register
call  register
```

We insert profiling code wherever this sequence of statements appears. It is possible that the same code could be generated for non-virtual function calls, however, we have not observed any such problems with the programs we have studied. The resulting profiled call looks like this:

```
add   register + offset, reserved_global_register
call  pic
load  [register + offset], register
call  register
```

We force the rest of the program to avoid using the `reserved_global_register` so that we can pass the profiling information in it. The runtime cost of profiling is quite large since it adds a statically bound call to every virtual function call site, so it is not a scheme which one would implement in a production C++ environment to record profile information.