Profile-Guided Receiver Class Prediction

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

Department of Computer Science and Engineering
Box 352530, University of Washington
Seattle, WA 98195-2350
{grove,jdean,garrett,chambers}@cs.washington.edu

Abstract

The use of dynamically-dispatched procedure calls is a key mechanism for writing extensible and flexible code in object-oriented languages. Unfortunately, dynamic dispatching imposes a runtime performance penalty. Some recent implementations of pure object-oriented languages have utilized profile-guided receiver class prediction to reduce this performance penalty, and some researchers have argued for applying receiver class prediction in hybrid languages like C++. We performed a detailed examination of the dynamic profiles of eight large object-oriented applications written in C++ and Cecil, determining that the receiver class distributions are *strongly peaked* and *stable* across both inputs and program versions through time. We describe techniques for gathering and manipulating profile information at varying degrees of precision, particularly in the presence of optimizations such as inlining. Our implementation of profile-guided receiver class prediction improves the performance of large Cecil applications by more than a factor of two over solely static optimizations.

1 Introduction

Object-oriented languages include dynamically-dispatched procedure calls (also known as messages or virtual function calls) as a key mechanism for making code more extensible and flexible. Dynamic dispatching introduces a level of indirection between client code and data type implementations that allows different and even unanticipated implementations to be manipulated by a single piece of client code without change. Unfortunately, dynamic dispatching imposes a run-time performance cost, due to both the direct overhead of dispatching the message and the indirect cost of lost opportunities for inlining and interprocedural analysis. In an effort to reduce the cost of object-oriented programming, some techniques have been devised to convert dynamic dispatches into statically-bound direct procedure calls when it turns out that the potential polymorphism is not being used by a particular piece of code; these techniques include intraprocedural and interprocedural static class analysis [Chambers & Ungar 90, Palsberg & Schwartzbach 91, Plevyak & Chien 94, Agesen & Hölzle 95] and class hierarchy analysis [Dean *et al.* 95, Fernandez 95]. However, some code really is polymorphic, manipulating objects of different implementations at different times, and so some form of dynamic dispatching appears to be required.

Appeared in *OOPSLA'95*, Austin, TX, October, 1995.

Even when code is polymorphic, it may be that one or two implementations are most frequent. For example, a drawing editor application may contain the following dynamically-dispatched `area` message sent to all the shapes in a scene:

```
Shape* s = ...;
a = s->area();    /* dynamically dispatched */
```

At run-time, many different implementations of the `Shape` abstract class may be sent the `area` message, but suppose that most of the time the receiver of the message is an instance of the `Rectangle` class. When one receiver class dominates, a message can be sped up on average by inserting an in-line run-time test for the dominant class followed by a branch to an optimized sequence of code if the test is successful:

```
Shape* s = ...;
if (s->class == Rectangle) {
    /* a statically-bound direct call */
    a = s->Rectangle::area();
} else {
    /* a dynamically-dispatched message */
    a = s->area();
}
```

Once the common case is separated and statically bound, additional optimizations such as inlining can be applied to further speed that case:

```
Shape* s = ...;
if (s->class == Rectangle) {
    /* inlined Rectangle::area */
    a = s->length * s->width;
} else {
    /* a dynamically-dispatched message */
    a = s->area();
}
```

Previous work has shown that this optimization, which we call *receiver class prediction*, can significantly improve performance. This optimization is particularly important in pure dynamically-typed object-oriented languages, where message sends are frequent and static information with which to optimize programs is scarce, but receiver class prediction can also be important in statically-typed hybrid languages: Calder and Grunwald argue how C++ [Stroustrup 91] programs can be sped up with this optimization, particularly on some modern processors where branch prediction hardware supports conditional branches much better than indirect procedure calls [Calder & Grunwald 94]. And even on hardware where indirect procedure calls are as fast as conditional branches, the indirect benefits of inlining and post-inlining optimization can be substantial.

To perform receiver class prediction at a particular call site, some sort of expected class frequency distribution is needed for that call site. The Deutsch-Schiffman Smalltalk-80 system [Deutsch & Schiffman 84] and the Self-89 [Chambers & Ungar 89] and Self-91 [Chambers 92] systems in effect included a small hard-wired table of expected classes for a few dozen common messages such as + and `if`. In the absence of better static information, these systems indexed into the prediction table using the name of the message as the key, and if a matching entry was found they performed receiver class prediction accordingly. The Self-93 system [Hölzle & Ungar 94] derives its predictions through a sophisticated on-line profiling and dynamic recompilation strategy, leading to a much larger prediction table derived from the application's run-time behavior and indexed by individual call site rather than message name. Based primarily on this technique, the Self-93 implementation achieved a speed roughly 40% that of optimized C++ for medium-sized applications, which was roughly 50% faster than previous Self compilers on the same applications.

Thus receiver class prediction, particularly prediction based on profiles of program execution, appears to have significant potential for optimizing object-oriented programs. However, several important questions remain unaddressed by previous work:

- At what *granularity* should profile data be maintained?
- What do profile-based receiver class distributions look like, across a range of hybrid and pure object-oriented languages and for different granularities? In particular, is it often the case that one receiver class is dominant at a call site, i.e., are receiver class distributions *strongly peaked*?
- Are profile-based receiver class distributions *stable* across different inputs, across a range of hybrid and pure object-oriented languages? If a program is optimized using a profile derived from one run, will that profile adequately predict future behavior of the program? Similarly, can a profile taken from an old version of a program be used effectively to drive receiver class prediction when compiling updated versions of the program? How does the granularity of profile information affect its stability?
- Can receiver class distributions derived from off-line profiling be exploited effectively? (The Self-93 system demonstrated that on-line profiling and recompilation is effective.) What kind of impact does this optimization have on bottom-line run-time performance? How do the different granularities of profile information affect run-time performance?
- How expensive is it to gather the necessary profile information? Is profile-guided optimization compatible with rapid program development environments? Can profile information be gathered from optimized programs? What is an efficient mechanism for representing and manipulating profile information at different granularities?

In this paper we address these questions. In the next section we present the *k*-CCP model for describing profile information of different granularities and describe implementation techniques for gathering off-line profile data and manipulating it internally within a compiler. In sections 3 and 4 we report that profiles of programs in C++ (a hybrid singly-dispatched statically-typed object-oriented language) and Cecil [Chambers 93] (a pure multiply-dispatched dynamically-typed object-oriented language) are strongly peaked and stable across inputs and even program versions. In section 5 we report that profile-guided receiver class prediction doubles the performance of large Cecil programs. Section 6 summarizes related work.

2 Implementing Profile-Guided Optimizations

2.1 Granularity of Profile Information

In off-line profiling, the run-time system monitors the receiver class distributions appearing for particular messages and/or call sites during program execution, saving the final distributions when the program completes. The compiler then uses these receiver class distributions to guide class prediction for subsequent compilations. The granularity at which receiver class distributions are recorded influences how effective receiver class prediction can be. If distributions are maintained only at the granularity of the names of the messages being sent, then all calls that send that message will have their distributions averaged together, producing a *message summary* distribution. For example, a single distribution will be recorded for the `area` message, derived by combining the distributions of all call sites that send the `area` message. If call sites sending a particular message have similar distributions, then a single message summary distribution is an adequate predictor. However, it is often the case that different call sites of a particular message have different distributions. For instance, one call site might send `area` only to rectangles and squares, while another call site might send `area` only to windows. If distributions are recorded at the granularity of individual call sites, producing *call-site-specific* distributions, the variations across call sites can be preserved and more precise predictions can be made. It is possible to make even more precise records of receiver class distributions by associating distributions with a stack of calling procedures enclosing a call site, up to some height of callers enclosing a call site. Such *call-chain-specific* distributions avoid blending together distributions from different clients of a method, such as with shared polymorphic library routines that are used differently by different parts of a program. In general, it probably would be more expensive to gather call-chain-specific distributions, but in the presence of inlining, call site distributions can be enhanced at no extra run-time cost to call chain distributions, for the chain of inlined calls enclosing the call site. However, information with more context has restricted applicability; in particular, message summaries can be useful for guiding class prediction for new code added to a program or even for completely different programs, while call-site-specific distributions only apply to previously-profiled call sites.

We have developed the *k*-CCP (Call Chain Profile) model for describing the granularity of information present in profiles, patterned after Shivers's *k*-CFA family of control flow analyses for Scheme [Shivers 88, Shivers 91]. A *k*-CCP receiver class distribution is associated with the message being sent and the specific call sites within the *k* procedures that dynamically enclose the call. A 0-CCP distribution is associated only with the message being sent, leading to a message summary distribution. A 1-CCP distribution is specific to a particular call site within a particular procedure, producing a call-site-specific distribution. Higher values of *k* reflect the additional context of a chain of *k* enclosing procedures. This same framework can be used to model other kinds of information about calls, such as the execution frequency of a call in particular contexts.

2.2 Gathering Profile Information

Profiling an application to get receiver class distributions requires that the program executable be built with the appropriate instrumentation. To enable long profiling runs and profiling of typical application usage, profiling should be as inexpensive as possible, since otherwise it may not be feasible to gather profile information. The expense of profiling and the ease with which different granularities of profile data can be gathered depends in large part on the run-time system's message dispatching mechanism. Some systems, including the implementations of Self and Cecil, use *polymorphic inline caches* (PICs) [Hölzle *et al.* 91]: call-site-specific association lists mapping individual receiver classes to target methods. To gather call-site-specific profile data, counter increments are added to each of the cases, and the counters for the PICs of all call sites are dumped to a file at the end of a program run. Each PIC is annotated with as much call chain context as is available after inlining. The run-time overhead of this profiling for our Cecil benchmarks (described in section 5) is 15-50%.

Other systems, such as C++, rely on dispatch tables to direct messages. To gather profiles of C++ programs, we have been using the stop-gap measure of recognizing the assembly code generated by our C++ compiler for a virtual function call, and inserting instrumenting code that records call-site-specific distribution information (not call chain information), at relatively high run-time cost. A realistic system needs to be developed for profiling programs written in C++ and other languages that use dispatch tables.

2.3 Profiling Optimized Code

To make profiling less intrusive, we wish to be able to profile optimized programs. In the presence of optimization, many dynamic dispatches that were present in the unoptimized program will no longer be performed, either because static analysis was able to determine the message target uniquely, or because receiver class prediction had been performed and the dynamic send had

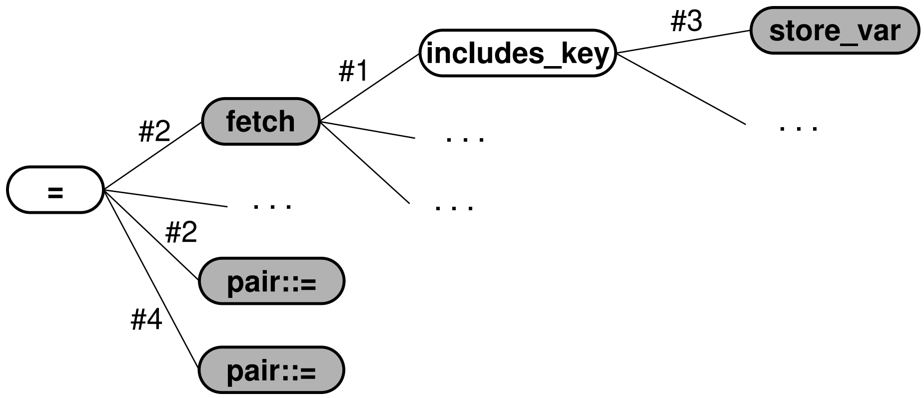

Figure 1: Call-Chain Representation

been avoided due to an earlier class test. If an optimized program were profiled naively, the resulting distributions would be quite misleading for use in further optimization. For example, for a send optimized through class prediction, the resulting profile would only include the *uncommon* receiver classes.

We use two techniques to support accurate and efficient profiling of optimized programs. First, we do not instrument sends that were statically-bound solely due to static analysis. We reason that such sends will likely be able to be optimized similarly without resorting to profile guidance, and so do not need to have data recorded for them in the profile database. Consequently, we avoid introducing the significant cost of instrumenting such sends. Second, for sends optimized through class prediction, we instrument the successful branches as well as the dynamic send, combining their results into a single distribution for the original unoptimized send. This preserves the high-frequency classes in the distribution accurately, for example allowing the compiler to detect when a common case becomes less common, but it can impose a significant performance cost. For example, in our system, `if` is implemented as a message to either `true` or `false`. This message is usually optimized through receiver class prediction, with "class" tests for `true` and `false` being inserted before the `if` message to catch the common cases. (If there are no other implementations of `if`, then the "otherwise" case is replaced by a call to the run-time system's error handler.) When profiling such a program, every `if` message will be instrumented to determine how many times `true` and `false` occurred. Such highly-accurate branch prediction information may be overkill for many systems. Counting successful `true` and `false` class tests accounts for most of the run-time overhead of profiling in our system; languages with more traditional control and data models may observe a far lower cost for gathering receiver class distributions in optimized programs.

If a program has been optimized with receiver class prediction, profiling the optimized program can lead to better profile information than the profile of the unoptimized program. Some call sites will have been optimized with receiver class prediction, leading to additional inlining, which produces new call sites with longer inlined call chains. The profile of the optimized program may have multiple separate receiver class distributions for a call site in a method that has been inlined in multiple places, while the profile of the original unoptimized program has only one blended distribution. This process can be repeated again, profiling the second program optimized with the profile of the first optimized program, and so on, until no more improvements occur in the profile (i.e., when no more inlining is performed). Section 5 reports on the impact this successive reprofiling has on run-time performance.

2.4 Representing Profile Information

In a given profile, there may be many different distributions associated with a given message, for different partially-overlapping call chains of varying lengths. Internally, we represent the profile information for a given message as a tree, where each node represents a particular call chain of length ≥ 0 and common prefixes of call chains have been factored. Each node may or may not have a receiver class distribution associated with it. For example, Figure 1 shows four receiver class distributions for the = message (shaded nodes have distribution information). The numbers along the arcs indicate the call site within the enclosing procedure. Two separate call-site-specific distributions are being recorded for the two distinct sends of = in the pair = method, a distribution is being recorded for the = send within the `fetch` method, and a distribution

Table 1: Benchmark Programs

Language	Program	Size (in lines)	Inputs
C++	`new` Self compiler	33,500	Small Self benchmarks, Cecil compiler written in Self
	`sic` Self compiler	14,900	Small Self benchmarks, Cecil compiler written in Self
	`doc` editor	15,400 + 24,900 library	Typing in one page of text. Randomly cutting and pasting in an existing 10 page document.
	`idraw` graphical editor	6,300 + 24,900 library	Drawing lots of rectangles. Exercising all possible shapes and text.
	Trace-driven memory subsystem simulator	22,000	Memory traces from the execution of the `gcc` and `doduc` programs
	teaching compiler for undergraduate course	4,600	Several small programs
Cecil	Global instruction scheduler for the MIPS	2,400 + 7,400 library	Several MIPS assembly files
	Cecil compiler in Cecil	38,000 + 7,400 library	Towers of Hanoi benchmark and compiler test suite, with and without optimization

is being maintained for the `fetch` method when it is inlined within the `includes_key` method that is itself inlined within the `store_var` method. It is likely that this latter dynamic occurrence of the = message will have a quite different (and more precise) receiver class distribution than the general distribution for = within `fetch`.

During compilation, when performing receiver class prediction for a given message send, we search the message's prediction tree, identifying the longest call chain prefix common to the call site and the tree. If that node has distribution information, we use it directly, assuming it to be the most accurate predictor of future call sites with that chain. If the node does not have profile data, it must have successor nodes (i.e., nodes indexed by longer call chains) with data, and a summary distribution for that node is calculated by summing the counts of each class of the successor distributions. For example, if a call site sending the = message is being compiled, but there is no profile data yet for that call site, then its call chain will share only the first node with the profile database's call tree for =. Since that node has no profile data of its own, all its successor nodes will be aggregated into one summary distribution for =, and cached in that node for future reference. In this fashion, global message summaries are calculated on demand from the more specific distribution information. Similarly, summaries at finer granularities can be calculated when needed.

3 Are Class Distributions Strongly Peaked?

Although receiver class prediction speeds up messages sent to predicted classes, it slows down messages sent to non-predicted classes. Therefore, in order for receiver class prediction to improve the performance of applications, receiver class distributions must be strongly peaked, i.e., have a small number of dominant classes. To determine how strongly peaked receiver class distributions are in practice, we examined the distributions generated by profiling the suite of large C++ and Cecil* programs described in Table 1. All of the applications were compiled using standard intraprocedural optimizations. For the C++ programs, we used `g++` version 2.3.3 with an optimization level of `-O2`. Because we are interested only in call sites that would be potential candidates for profile-guided class prediction, we instrumented virtual function calls only. We did not measure the class distributions for non-virtual member functions, since these call sites are already statically bound and consequently have no need for class prediction. The Cecil programs were compiled using

* Cecil is a multiply dispatched language, so many call sites do not have a single receiver class. The analog of receiver class prediction for a multi-method-based language tests the classes of several arguments. For the purposes of this paper, we treat each such tuple of classes as a single "receiver class."

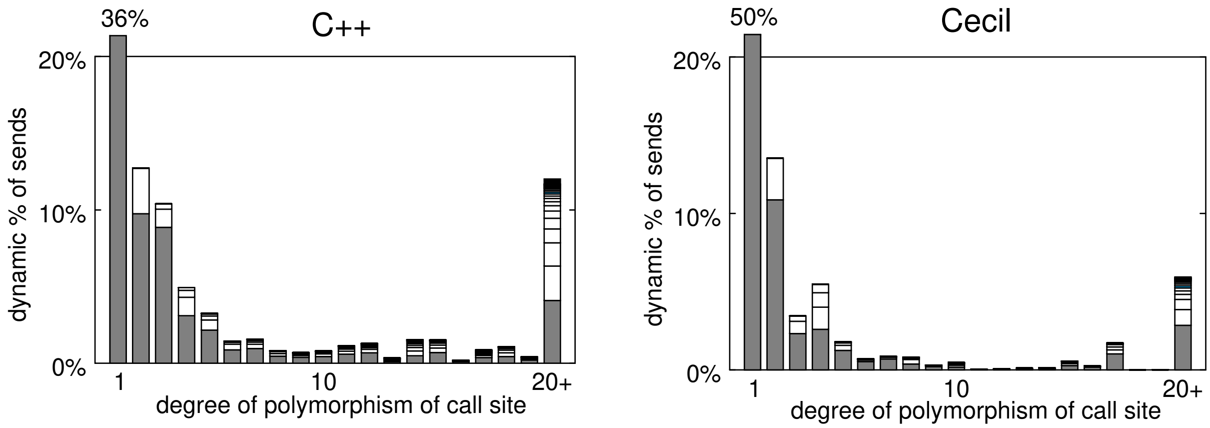

Figure 2: Receiver Class Polymorphism

hard-wired class prediction for common messages, iterative intraprocedural class analysis, splitting, inlining, and standard intraprocedural optimizations. This default level of optimization was chosen to compensate for the pure object-oriented model and user-defined control structures in Cecil, eliminating dynamic dispatches due to simple arithmetic operations like + and control structures like `if` that are not present in languages like C++ which have a selection of built-in data types and control structures.

For each language, we computed call-site-specific (1-CCP) class distributions and merged together call sites that exhibited the same degree of polymorphism (i.e., those that had the same number of distinct receiver classes at runtime). The results are shown in Figure 2. The height of a bar in each histogram indicates the dynamic percentage of sends (virtual function calls) in the benchmarks with that degree of polymorphism. Each bar of the histogram, reporting execution frequencies of sends with degree *N* polymorphism, is vertically divided into *N* parts showing the relative frequency from the most common to the least common receiver. Thus, the bottom shaded portions of the bars report the frequency with which messages were sent to the most common receiver class at that call site. The sends with polymorphism greater than 20 are collected together into the 20+ bin. These graphs show the aggregate results for all of the benchmarks in a particular language with each benchmark program weighted equally. (The same graphs drawn for each program in isolation have similar shapes.)

In both languages, the most common receiver class at a call site receives the majority of the messages sent at that call site; 71% of the C++ messages and 72% of the Cecil messages were sent to the most common receiver class. In C++, 36% of the dynamic dispatches occurred at call sites which only had a single receiver class, and 50% of the Cecil messages were sent at call sites with a single receiver class. Calder and Grunwald also studied the characteristics of class distributions of C++ programs and found that 91% of the messages were sent to the most common receiver class and that 66% of the call sites only sent to a single receiver class. Our C++ programs are larger, and judging from the greater degree of polymorphism, programmed in a more object-oriented style. However, there are still a large percentage of virtual function call sites with a single receiver class. In summary, we find that class distributions in both C++ and Cecil programs are strongly peaked, suggesting that both languages are promising candidates for receiver class prediction.

4 Are Class Distributions Stable?

Even if we have a program with very strongly peaked 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 future runs. We use the term *stability* to refer to the degree that one program run's profile data predicts the behavior of another program run. Stability can be measured by comparing the shape of the class distributions generated from two profiles. We identify three important kinds of stability:

- *cross input*: How stable are the class distributions generated by running the same program on two different input sets? Without this kind of stability, effective profile-guided class prediction is impossible.
- *cross version*: How stable are the class distributions generated by running two different versions of the same program on the same input set? This kind of stability would facilitate the usage of profile-guided class prediction in a program development environment since it would allow the costs of profiling

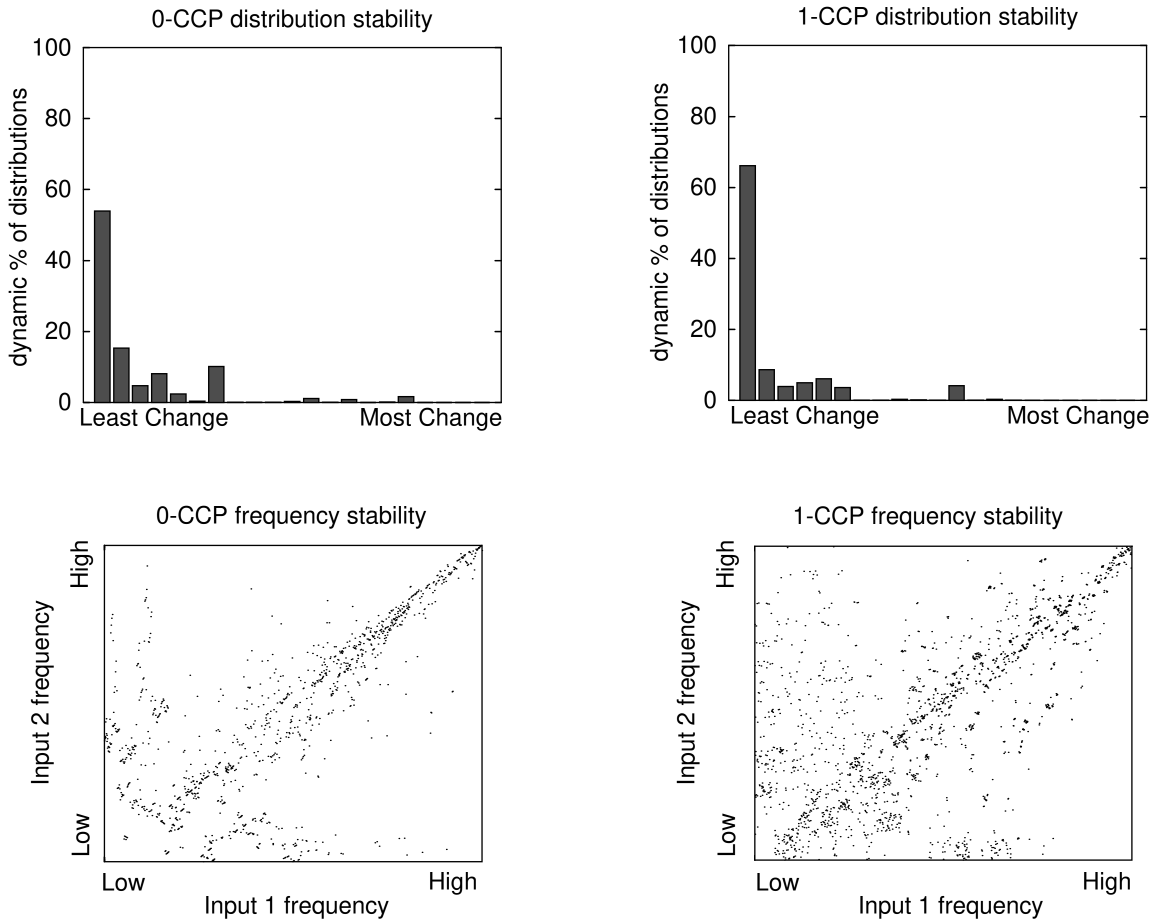

Figure 3: C++ Stability Across Inputs

to be amortized across many edit-compile-debug cycles.

- *cross program*: How stable are the class distributions of call sites located in a library shared by two different programs? This kind of stability would enable the use of profile-guided class prediction to build a single, shared optimized version of the library or to initialize the profile database for a fresh application without profile data of its own.

In addition to the stability of class distributions, we are also interested in assessing the stability of profile-derived execution frequencies. Since our compiler uses this information to decide where to apply class prediction and to guide inlining decisions, we would like this information to be fairly stable.

Section 4.1 defines the metrics we use to measure stability. Section 4.2 describes our study of cross-input stability. We then consider cross-version stability in section 4.3 and cross-program stability in section 4.4. Section 4.5 compares off-line to on-line profiling, in light of these stability results.

4.1 Stability Metrics

We use several metrics to evaluate the stability of receiver class distributions. One metric is the L_2 difference* between two normalized distributions, which is a very good indicator of high or low stability, but is not a particularly accurate metric for assessing two distributions that are somewhat similar. One advantage of this metric is that it allows an abstract comparison of two class distributions that is independent of any application of the distributions. We also use two additional metrics that are more directly related to our intended application of the class distributions. The FirstSame metric classifies two distributions as the same as long as their most common

* 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.

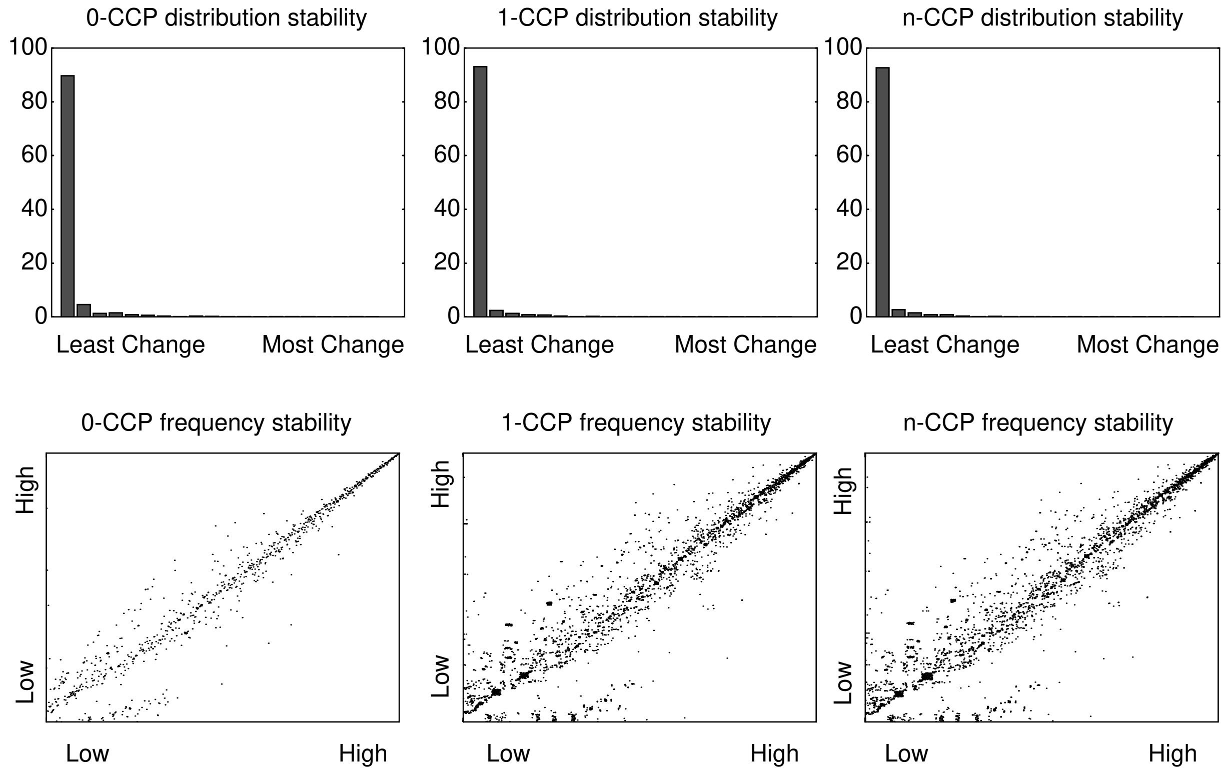

Figure 4: Cecil Stability Across Inputs

receiver classes are the same. Since the most common receiver class is usually the most important for class prediction, we expect that the FirstSame metric is a fairly realistic measure of stability for the purpose of guiding receiver class prediction. We also use a final, very stringent metric, OrderSame, which classifies two distributions as the same only if they are comprised of the same classes in the same frequency order.

We illustrate the stability of profile-derived execution frequencies by drawing a scatter plot in which the x coordinate is a call site's position in the sorted order of frequencies in one profile and the y coordinate is its position in the sorted order of frequencies of the second profile. The more closely correlated the two sortings are, i.e., the more closely the points lie to a 45° line, the more stable the execution frequency is. Points along the x or y axis correspond to messages (call sites) that were executed in only one of the inputs.

4.2 Stability Across Inputs

To assess the cross-input stability of class distributions, we compared the profiles of each of our C++ and Cecil programs run on two different input sets. We attempted to make the inputs as different as possible to present a worst case scenario for the cross-input stability of receiver class distributions. We first present the results of our two interactive C++ programs, `doc` and `idraw`, which are the least stable of all the measured applications. We only present 0-CCP and 1-CCP stability for these programs, since our C++ profiling technology does not support gathering n-CCP distributions. We then present the stability results for the two Cecil applications, `compiler` and `instr-sched`. These two programs, and the remainder of the C++ programs, are batch-oriented applications and have similar cross-input stability characteristics.

When we measured the interactive InterViews programs, `doc` and `idraw`, we attempted to make the inputs fairly distinct. 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, including rectangles, lines, ellipses, and text, and the usual transformation operations, such as 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 3. The receiver class profiles, and the execution frequencies to some extent, of these interactive C++ programs are fairly stable across inputs.

Receiver class profiles of the Cecil applications and of the batch-oriented C++ applications were even more stable than the InterViews C++ programs. Our results for the Cecil applications, averaged over both programs, are shown in Figure 4. n-CCP results refer to profiles including call-chain-specific distributions, where the length of each chain is determined by the amount of inlining enclosing the call site. Again, we tried to generate different inputs, for instance by running the `compiler` benchmark on different input programs with different optimization settings.

Table 2 presents the results of applying the FirstSame and OrderSame metrics to measure cross-input stability. The numbers represent the dynamic percentage of class distributions considered the same by the metric in question.

Table 2: Cross-Input Stability Summary
% of distributions considered equivalent

	C++	**Cecil**
0-CCP, FirstSame	99%	99.9%
1-CCP, FirstSame	79%	99.9%
n-CCP, FirstSame	N/A	99.9%
0-CCP, OrderSame	28%	84%
1-CCP, OrderSame	45%	87%
n-CCP, OrderSame	N/A	88%

According to the FirstSame metric, class distributions were extremely stable across inputs. Even by the stringent OrderSame metric the Cecil distributions were very stable and the C++ distributions were somewhat stable. Since we believe that the FirstSame metric closely models how profile-guided class prediction uses receiver class distributions, this data indicates that distributions have enough cross-input stability to support this optimization.

4.3 Stability Across Program Versions

We would like class distributions to be stable across versions of a program undergoing rapid development. If this stability holds, then a profile from an older version of the program can be reused over many future versions without requiring reprofiling after each programming change. To determine the degree of cross-version stability exhibited by class distributions, we used RCS logs [Tichy 85] to recreate 12 snapshots in the development of the Cecil compiler. The measured period began approximately one month after the compiler reached a basic, stable level of functionality (the compiler could optimize itself) with snapshots being taken twice a month for a period of six months. During this time, the application almost doubled in size, growing from 22,000 to 38,000 lines as additional optimization passes and other functionality were added. Support for selective recompilation was added, which had implications for many pieces of the compiler. Also during this period, many existing pieces of the compiler were substantially modified. The class hierarchies defining its internal representation were completely redesigned and a new iterative data flow analysis engine was added; many key data structures such as the method table and the compile-time method lookup cache were replaced with more efficient implementations. There were also pieces of the compiler that remained largely unchanged during this period; for example, only slight changes were made to the scanner and parser.

The results of our cross-version stability study are presented in Figure 5. Only 0-CCP and 1-CCP distributions are reported, since older versions of the system did not support collecting n-CCP distributions. As in the previous section, all data points represent dynamic frequencies. We compared each profile to every later profile, and plotted lines showing how each profile degraded in predictive quality over time. For example, the data points whose *x* coordinates are “1 month” represent the comparison of two profiles which were taken one month apart. There are 12 lines in each graph each one representing a single profile compared to all later profiles. According to the very stringent OrderSame metric, the distributions were somewhat unstable, since a significant percentage of the distributions changed in only a two-week period. However, according to the more realistic FirstSame metric, class distributions were quite stable. Applying this metric, fewer than 5% of the 0-CCP distributions changed over the entire six-month period, and it took around two months before more than 10% of the 1-CCP distributions changed. This suggests that old profiles are still quite useful for guiding receiver class prediction for future versions of a program.

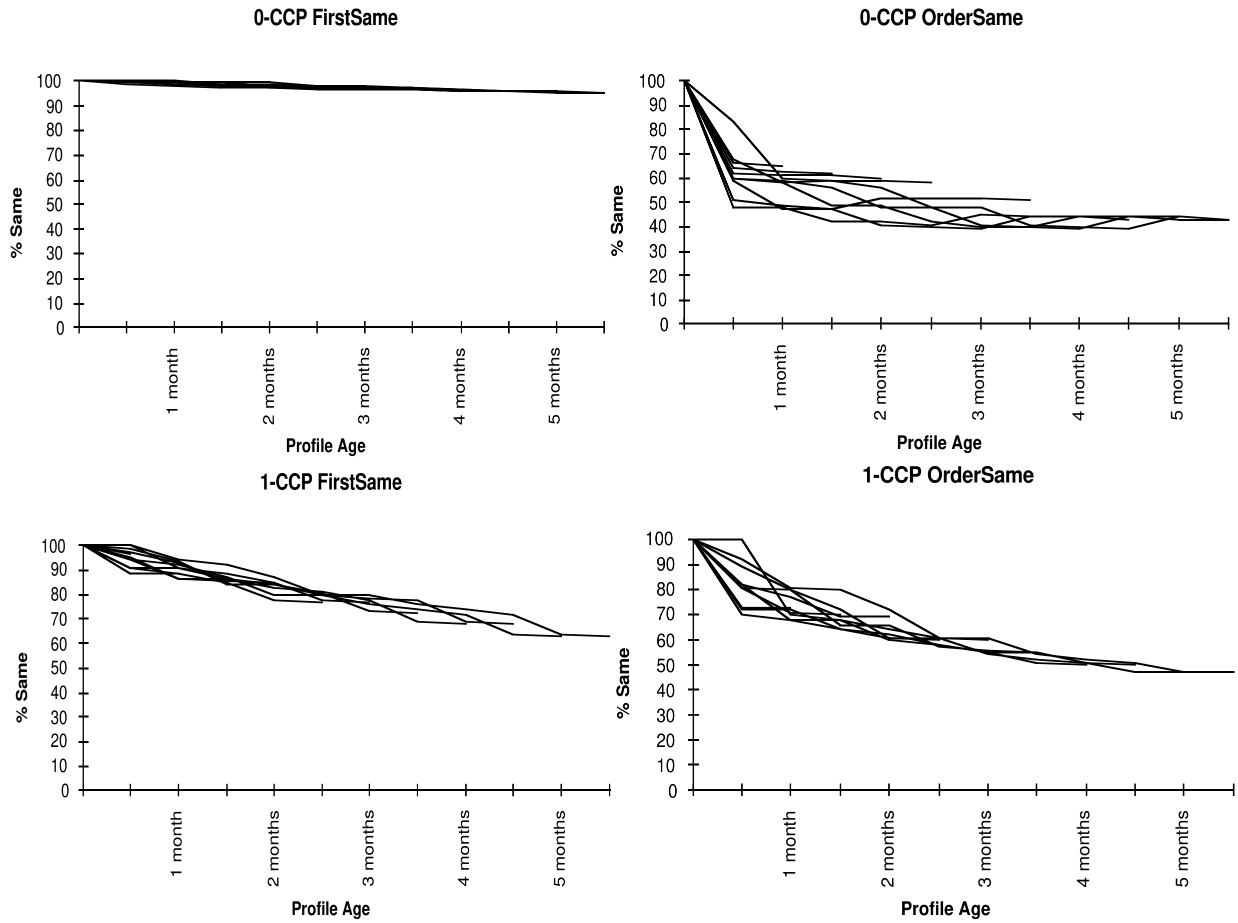

Figure 5: Cecil Stability Across Versions

Current work includes gathering similar revision histories for C++ applications to confirm the Cecil results.

We also examined the cross-version stability of execution frequencies. The two scatter plots in Figure 6 show the cross-version stability of 1-CCP execution frequencies after a period of one and six months. Only call sites that appear in both profiles are plotted, thus call sites that were added between the first and the second profile do not appear. These scatter plots reveal that execution frequencies remain fairly stable over a one-month period and somewhat stable even over a six-month period.

4.4 Stability Across Programs

Given that different programs frequently share pieces of code in common libraries, it would be useful to examine cross-program stability of distributions in the common portion of different programs. Unfortunately, we currently do not have a sufficiently broad range of programs that all make use of shared code to make any conclusive statements on the stability of cross-program distributions. We plan to experiment with combining the class distributions of several clients of a library to produce an aggregated profile of the library. This aggregated profile would allow compilation of a single, shared optimized version of the library that reflected common usage patterns for a suite of applications. If profiles are relatively stable across programs, then this library would be reasonably optimized even for programs not included in the initial profiling suite. We are in the process of collecting additional applications and hope to experiment with aggregating distributions from multiple programs in the future.

4.5 On-Line vs. Off-Line Profiling

Since we are focusing on off-line profiling, we are assessing the stability of a class distributions derived from the entire run of the program. One potential advantage of an on-line approach is that it allows the system to adapt to changes in a program's behavior within a single run, for instance as the program goes through different phases in its

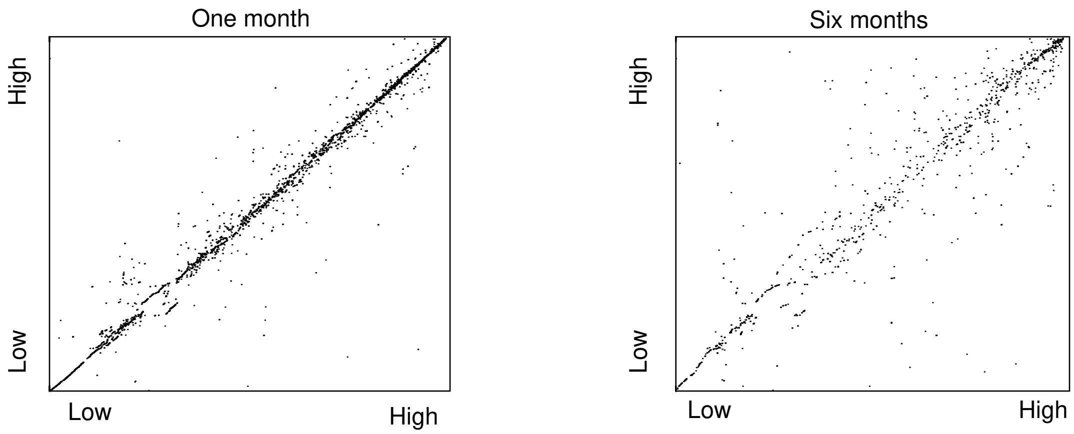

Figure 6: 1-CCP Cross-Version Execution Frequencies

execution. However, our results suggest that cross-phase instability is not a serious problem for the off-line approach, at least for our benchmark programs. Since 70% or more of dynamic messages are sent to the most common receiver class (see Figure 2), an on-line system could hope to improve only the remaining 30% of the calls. The benefits an on-line system would receive by reordering the class tests and re-optimizing common cases are likely to be relatively small, and these benefits would be offset by the additional run-time costs of monitoring class distributions on-line and doing run-time recompilation.

5 Can Class Distributions Be Exploited?

In the previous sections we saw that class distributions were both strongly peaked and quite stable across both inputs and versions. In this section we examine the degree to which these abstract qualities of the profiles translate into improved execution performance. We implemented receiver class prediction in our Cecil compiler, and examined its effectiveness on five medium-to-large Cecil programs, described in table 3.

Table 3: Cecil Benchmarks

Program	**Lines**[a]	**Description**
richards	400	Operating system simulation
deltablue	650	Incremental constraint solver
instr-sched	2,400	MIPS global instruction scheduler
typechecker	17,000	Cecil typechecker
compiler	38,000	Cecil compiler

a. Plus 7,400-line standard library.

The applications were compiled with the following levels of optimization:

- ***unopt***: No message-level optimizations performed.
- ***std***: Standard static optimizations, including iterative intraprocedural class analysis, class hierarchy analysis, inlining, hard-wired class prediction for a small set of common messages, splitting, constant propagation and folding, common subexpression elimination, dead assignment elimination, and closure optimizations.
- ***0-CCP***: Standard augmented by profile-guided class prediction using 0-CCP (message summary) class distributions generated by profiling a ***std*** optimized version of the program.
- ***1-CCP***: Standard augmented by profile-guided class prediction using 1-CCP (call-site-specific) class distributions generated by profiling a ***std*** optimized version of the program.
- ***n-CCP0***: Standard augmented by profile-guided class prediction using n-CCP (call-chain) class distributions generated by profiling a ***std*** optimized version of the program.
- ***n-CCP1***: Standard augmented by profile-guided class prediction using n-CCP (call-chain) class distributions generated by profiling a ***n-CCP0*** optimized version of the program.
- ***n-CCP2***: Standard augmented by profile-guided class prediction using n-CCP (call-chain) class distributions generated by profiling a ***n-CCP1*** optimized version of the program.

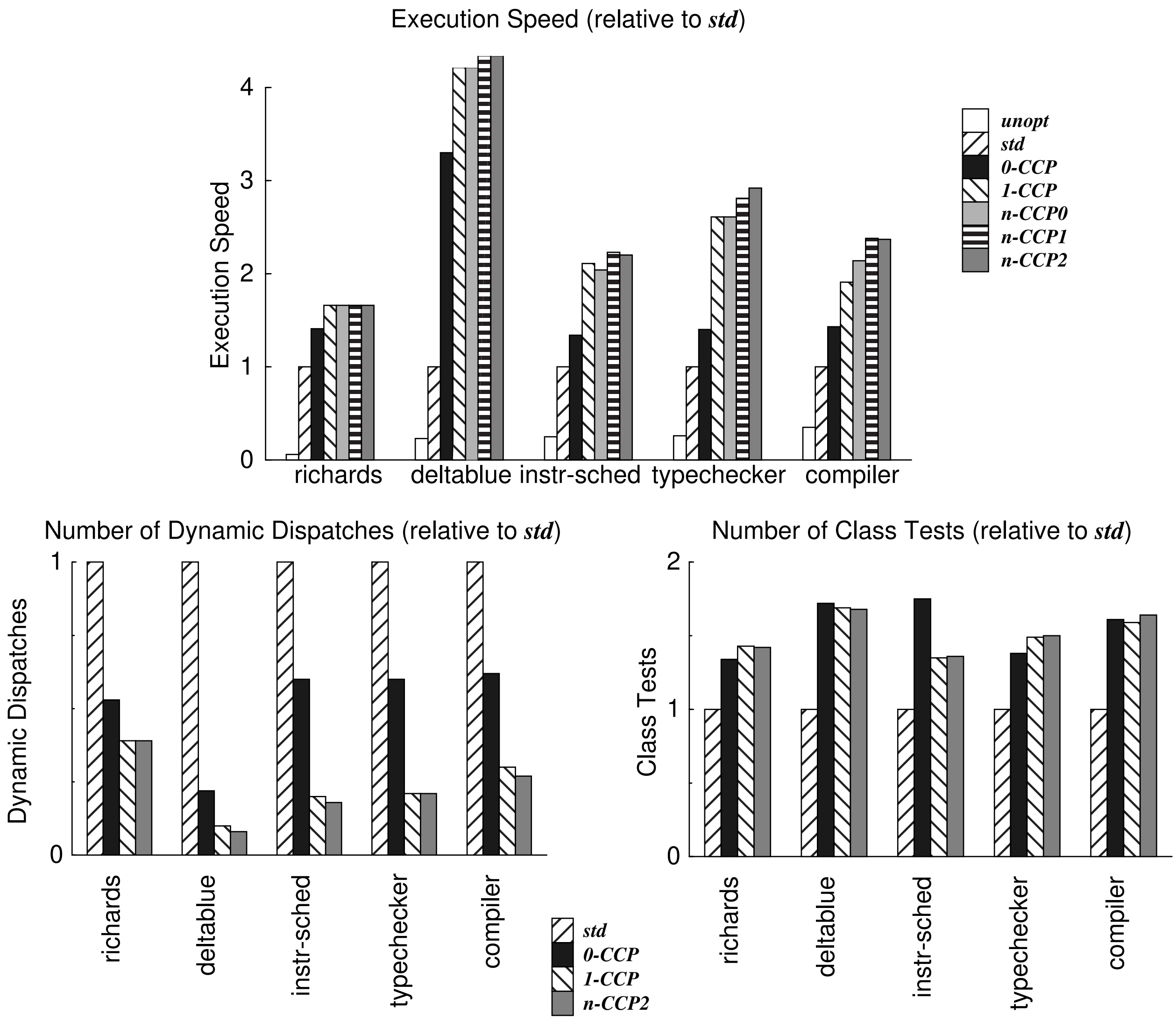

Figure 7: Effectiveness of Profile-Guided Receiver Class Prediction for Cecil

For these experiments, our compiler generated portable C code, which we then ran through `gcc -O2` to complete the compilation. We ran our experiments on a lightly-loaded SparcStation-20/61 with 128MB of main memory.

Figure 7 shows the execution speeds of the benchmarks compiled with each of these compiler configurations. (Appendix A reports the raw data.) Speeds were normalized to that of the ***std*** version of each benchmark. All of the programs (except `richards` which takes no input) were run with different inputs than those used to generate the profiles used to optimize them. Differences in execution speed of less than 5% are probably not significant, due to the direct-mapped instruction cache organization on the Sparc.

Profile-guided receiver class prediction resulted in speed-ups of a factor of roughly two over purely static optimizations. The more precise information available in the 1-CCP distributions allowed improvements of 18-86% over 0-CCP distributions, allowing 1-CCP-based class prediction to obtain the majority of the performance benefits obtained by the more sophisticated n-CCP configurations. However, the increased precision of the n-CCP distributions enabled additional gains of up to 24% compared to a program optimized with 1-CCP distributions, and the largest differences were for the two largest benchmark programs, suggesting that n-CCP information becomes more useful as programs become larger. The different n-CCP distributions capture the effect

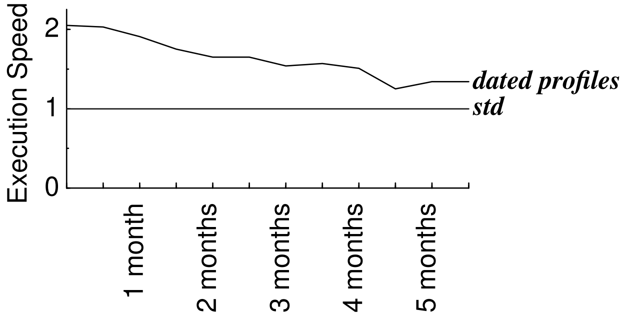

Figure 8: Effectiveness of Receiver Class Prediction Using Old Profiles

of repeatedly profiling and reoptimizing a program, lengthening the call chains of some call sites' receiver class distributions in the process. For these programs, the average call chain depth was around 1.5 (i.e., somewhat longer on average than call-site-specific profiling), with the largest benchmark, `compiler`, having an average depth of 1.7 and a maximum depth of 9. The n-CCP profiles stabilized quite quickly, after 1 to 3 iterations.

A less implementation-dependent measure of the effectiveness of profile-guided receiver class prediction is the percentage of dynamic dispatches it eliminates and the number of additional receiver class tests executed to avoid these dispatches. The second and third graphs in Figure 7 show the number of executed dynamic dispatches and in-line class tests for each benchmark, normalized to the execution counts of ***std*** (***std*** class tests are the result of hard-wired class prediction for common messages). These graphs illustrate that as more call chain context is available in the profile, more dynamic dispatches can be eliminated. Furthermore, because of the increased precision of the profile data provided by longer call chains, this decrease in number of dynamic dispatches comes without a substantial increase in the number of class tests inserted.

To confirm our supposition in section 4.3 that there was sufficient cross-version stability to allow the use of old profiles to optimize newer versions of the same program, we used the 1-CCP distributions from each of the profiled dates to optimize the `compiler` benchmark program sources from the most recent date. Figure 8 plots the execution speeds of the `compiler` programs compiled with profile data of varying ages (***dated profiles***) against that of a `compiler` program optimized with no profile data (***std***). Clearly, even using fairly inaccurate profile information is much better than using no profile data at all. Also, it appears that profiling once every 4 to 6 weeks would have been sufficient to maintain execution speeds that were within 90% of those enabled by completely up-to-date profiles.

These results demonstrate that off-line profile-guided class prediction can improve the performance of applications written in pure object-oriented languages, complementing the earlier results for on-line profiling in the Self-93 system. We are currently working on adding C++ and Modula-3 [Nelson 91] front-ends to our compiler so we can directly assess the impact of profile-guided class prediction on statically-typed hybrid object-oriented languages.

6 Related Work

Smalltalk-80 [Deutsch & Schiffman 84], Self-89 [Chambers & Ungar 89], and Self-91 [Chambers 92] compilers have all utilized hard-wired receiver class prediction (called *type prediction* in the Self work) to eliminate much of the overhead due to their pure object model and user-defined control structures. The Self-93 [Hölzle & Ungar 94] work demonstrated that on-line call-site-specific profile-based class prediction (called *type feedback* in that work) can substantially improve the performance of Self programs, but it did not investigate the effectiveness of off-line profiling, demonstrate the applicability to hybrid languages such as C++, or examine the peakedness or stability of class distributions.

Calder and Grunwald considered several ways of optimizing dynamically-bound calls in C++ [Calder & Grunwald 94]. They examined some characteristics of the class distributions of several C++ programs and found that although the potential polymorphism was high, the distributions seen at individual call sites were strongly peaked. Our C++ results confirm this general result, but the programs we measured are significantly larger and appear to make heavier use of object-oriented programming techniques, such as deep class hierarchies and factoring, judging by the greater degree of observed polymorphism.

Wall [Wall 91] investigated how well profiles predict relatively fine-grained properties such as the execution frequencies of call sites, basic blocks, and references to global variables. He reported that profiles from actual runs of a program are better indicators than static estimates but still are often inaccurate. We investigated the predictive power of a coarser-grained kind of information, receiver class distributions, and found that they have much greater stability. We also examined stability across program versions, a property not considered by Wall.

The IMPACT C compiler [Chang *et al.* 92] uses profile-derived execution frequencies to guide inlining. They report a 10% improvement in the speed of C programs using one set of inputs to predict behavior on another set of inputs. Thus, they demonstrate that the cross-input stability of their profiles was sufficient to yield performance benefits.

7 Conclusions

Previous work has argued that receiver class prediction can be effective at optimizing pure and hybrid object-oriented languages. In this work we sought to ground this optimization on a solid foundation by studying the properties of the dynamic profiles which drive receiver class prediction. We have analyzed profiles from large C++ and Cecil programs and determined that they are both strongly peaked and stable across inputs. We identified that cross-version stability is also important to the practicality of day-to-day use of profile-guided class prediction in a program development environment, and we determined that receiver class distributions exhibit significant cross-version stability. We verified that off-line profile-guided receiver class prediction is an effective optimization for Cecil programs, speeding large programs by a factor of two. We introduced the *k*-CCP model for describing the precision of profile information, described an implementation strategy for manipulating collections of call chain distributions, demonstrated that the additional context of call chains over call sites leads to less polymorphic, more peaked, and more stable receiver class profiles, and that indeed these properties lead to more effective profile-guided receiver class prediction. We described a method for profiling optimized code accurately and reasonably efficiently. Current work includes extending some of the experiments to include C++ and other languages, in particular the cross-version stability studies and the execution performance studies, expanding our benchmark suite, and developing efficient profiling methods for languages like C++ that use dispatch tables.

We have been using profile-guided class prediction since the spring of 1994 during our day-to-day development of the Cecil compiler itself. Our programming environment maintains a persistent internal database of profile information that is automatically consulted by the compiler for each compilation. Except for periodically gathering new profile data, exploiting profile data to guide receiver class prediction is virtually transparent to the Cecil programmer.

Acknowledgments

We thank Urs Hölzle and Karel Dreisen for comments on earlier drafts of this paper. This research is supported in part by a NSF Research Initiation Award (contract number CCR-9210990), a NSF Young Investigator Award (contract number CCR-9457767), a grant from the Office of Naval Research (contract number N00014-94-1-1136), and gifts from Sun Microsystems, IBM, Pure Software, and Edison Design Group.

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

References

[Agesen & Hölzle 95] Ole Agesen and Urs Hölzle. Type Feedback vs. Concrete Type Analysis: A Comparison of Optimization Techniques for Object-Oriented Languages. In *Proceedings OOPSLA '95*, Austin, Tx, October 1995.

[Calder & Grunwald 94] Brad Calder and Dirk Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 397–408, Portland, Oregon, January 1994.

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

[Chambers & Ungar 90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. *SIGPLAN Notices*, 25(6):150–164, June 1990. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.

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

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

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

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

[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*, pages 297–302, Salt Lake City, Utah, January 1984.

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

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

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

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

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

[Shivers 88] Olin Shivers. Control-Flow Analysis in Scheme. *SIGPLAN Notices*, 23(7):164–174, July 1988. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.

[Shivers 91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.

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

[Tichy 85] Walter F. Tichy. RCS-A System for Version Control. *Software Practice and Experience*, 15(7):637–654, July 1985.

[Wall 91] David W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. *SIGPLAN Notices*, 26(6):59–70, June 1991. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.

Appendix A Raw Data

Execution times are the median time in seconds from 9 runs on a SPARC-20/61 with 128 MB of memory

Table 4: Execution Times (seconds)

Configuration	Richards	Deltablue	InstSched	Typecheck	Compiler
unopt	14.350	2.860	23.370	293	1,989
std	0.790	0.660	8.450	77	697
0-CCP	0.410	0.200	4.190	52	489
1-CCP	0.380	0.150	3.610	30	365
n-CCP0	0.370	0.150	3.640	30	325
n-CCP1	0.370	0.140	3.200	28	293
n-CCP2	0.370	0.140	3.430	27	294

Table 5: Dynamically Dispatched Message Sends (×1000)

Configuration	Richards	Deltablue	InstSched	Typecheck	Compiler
unopt	12,506	1,248	9,821	84,952	435,401
std	1,583	351	2,889	17,411	104,096
0-CCP	840	77	1,747	10,364	64,254
1-CCP	623	34	580	3,667	31,220
n-CCP0	619	27	580	4,090	30,459
n-CCP1	619	26	534	4,091	30,069
n-CCP2	619	26	534	3,647	27,744

Table 6: Executed Class Tests (×1000)

Configuration	Richards	Deltablue	InstSched	Typecheck	Compiler
unopt	0	0	0	0	0
std	2,147	172	1,722	10,590	48,673
0-CCP	2,882	296	3,020	14,613	78,491
1-CCP	3,073	291	2,329	15,808	77,405
n-CCP0	3,050	289	2,329	15,500	77,989
n-CCP1	3,050	289	2,339	16,290	82,145
n-CCP2	3,050	289	2,339	15,856	79,701