Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages

Ole Agesen

Urs Hölzle†

Abstract: Two promising optimization techniques for object-oriented languages are type feedback (dynamic) and concrete type inference (static). We directly compare the two techniques, evaluating their effectiveness on a suite of 23 SELF programs while keeping other factors constant.

Our results show that both systems inline >95% of all sends and deliver similar overall performance with one exception: SELF's automatic coercion of machine integers to arbitrary-precision integers upon overflow confounds type inference and slows down arithmetic-intensive benchmarks.

We discuss several other issues which, given the comparable run-time performance, may influence the choice between type feedback and type inference.

1. Introduction

The dynamic dispatch present in object-oriented languages impairs many static code analysis and optimization techniques because they rely on statically knowing a program's call graph. Thus, calls not only slow down the program through the calling overhead per se but also through optimization opportunities destroyed by dynamically-dispatched calls. To make matters even worse, object-oriented programs tend to contain more calls at the source level than procedural programs since the object-oriented programming style encourages factoring code into small pieces to obtain fine-grained reuse. Because traditional compilers are unable to remove message sends, object-oriented programs usually exhibit a higher calling frequency than procedural programs. The frequent calls, combined with the scarce opportunities for traditional code optimizations, can lead to poor run-time performance.

Thus, the key to efficient implementation of object-oriented languages is to eliminate dynamically-dispatched calls by statically binding or inlining them. However, to inline a dynamically-dispatched call, the compiler must know the *exact* class of the receiver object. Unfortunately, this class is often unknown, even in statically-typed languages. Consider the following C++ code fragment:

```
GraphicalObject* obj;
...
obj->moveTo(0, 0);
```

Despite the type declaration, a C++ compiler cannot statically bind the `moveTo` call because it does not know the object's exact class (e.g., whether it is an instance of class `Point` or class `Rectangle`). Recently, however, two techniques have emerged which promise to enable better optimization of dynamically-dispatched calls by providing the compiler with precise information about the class of the receiver:

- *Type feedback* monitors previous executions of the program to determine the set possible receiver classes, and
- *Concrete type inference* computes the set of possible receiver classes by analyzing the program's source code.

† Authors' addresses: Ole Agesen, Computer Science Department, Stanford University, Stanford, CA 94305; agesen@cs.stanford.edu. Urs Hölzle, Computer Science Department, University of California, Santa Barbara, CA 93106; urs@cs.ucsb.edu.

Technical Report TRCS 95-04, Department of Computer Science, University of California, Santa Barbara, March 1995. UCSB Computer Science technical reports are available via ftp from ftp.cs.ucsb.edu or via WWW from http://www.cs.ucsb.edu/TRs.

A priori, each technique can potentially outperform the other in its ability to support optimization of object-oriented programs:

- Type feedback may generate better code because it takes into account the relative frequencies of receiver classes rather than treating them all as equally likely.
- Type inference may generate better code because it can completely eliminate dispatch for many message sends.

Previous studies have reported the effectiveness of type inference and type feedback (e.g., [PC94a] and [HU94a]) but direct comparisons have been impossible because important other factors were different, including programming language, compiler technology, and choice of benchmarks. The main contribution of this paper is a detailed comparison which is:

- *direct*, because we have been able to connect both type feedback and type inference to the same compiler back end, use the same run-time system, and execute the same suite of benchmarks in both cases; and
- *realistic*, because both the type feedback system and the type inferencer represent high-quality implementations of these concepts (the underlying SELF-93 system has been shown to significantly outperform commercial Smalltalk implementations [HU94a]).

In the remainder of this paper, we briefly summarize the two techniques (section 2), quantitatively compare them step by step (section 3), and qualitatively discuss their relative strengths and weaknesses (section 4). We then review related work (section 5) and finally offer our conclusions (section 6).

2. Background

2.1 SELF

SELF [US87] is a *pure object-oriented language*: all data are objects, and all computation is performed via dynamically-bound message sends (including all instance variable accesses, integer arithmetic, and control structures like `if` and `while`). SELF merges state and behavior: syntactically, method invocation and variable access are indistinguishable—the sender of a message does not know whether the message is implemented as a simple data access or as a method. Consequently, all code is *representation independent* since the same code can be reused with objects of different structure, as long as those objects correctly implement the expected message protocol. SELF's pure semantics result in very frequent message sends; in this respect, it is even harder to implement efficiently than Smalltalk.

2.2 Terminology

The term "type" commonly refers to several distinct concepts, such as abstract types (interfaces), concrete types (implementations), or sets of classes. To avoid possible confusion, we will use the following terminology throughout this paper:

- A *class* is a data structure that exactly describes the implementation of its instances, i.e., their size, layout, and the implementations of all methods defined for that class. Each object has exactly one class, and all instances of a class share the same implementation.

  Classes need not be visible at the language level. For example, SELF, the language used in the study, is prototype-based and thus has no notion of classes in the language. Nevertheless, the implementation maintains internal descriptors to keep track of each object's layout and methods, and these descriptors are equivalent to what we term "class" here.
- A *type* is a set of classes, possibly including the unknown class. Thus, a type like `{Point}` denotes "an object of class Point" whereas `{Point, unknown}` denotes "an object of class Point or any other class". The latter kind of type, though theoretically equivalent to `{unknown}`, is used by type feedback for reasons that will become clear shortly.

The remainder of this section briefly reviews type feedback and type inference; both systems have been described in more detail elsewhere ([Age94a], [Age95], [HU94a], [Höl94]).

2.3 Type feedback

The key idea of type feedback is to extract type information from previous executions and feed it back to the compiler (Figure 1). This feedback can happen dynamically (i.e., while the program is running) or statically (after execution

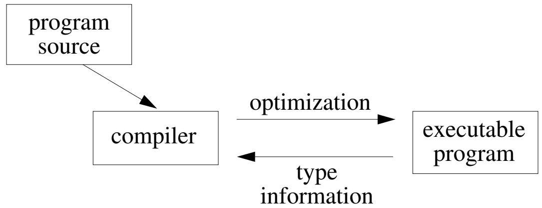

Figure 1. Overview of Type Feedback

With type feedback, the optimizing compiler uses "type profile" information gleaned from previous executions to specialize code to the frequently occurring receiver classes.

completed, as in traditional profile-based optimization). Type feedback uses an instrumented version of a program to record the program's *type profile*, i.e., a list of receiver classes (and, optionally, their frequencies) for every single call site in the program. To obtain the type profile, the standard method dispatch mechanism is extended in some way to record the desired information, e.g., by keeping a table of receiver classes per call site.

Based on the type feedback information, the compiler can predict likely receiver classes. For example, if type feedback indicates that `obj`'s class always was `Point`, the compiler could transform the call `obj->moveTo(0, 0)` into the following code:

```
if (obj->class == #Point) {
   /* inlined copy of Point::moveTo */
   obj->x = obj->y = 0;
} else {
   /* handle non-Point case here */
}
```

For `Point` receivers, the above code sequence will execute significantly faster since the original virtual function call is reduced to a simple load instruction and a comparison. Inlining the `moveTo` method not only eliminates the calling overhead but also enables the compiler to optimize the inlined code using dataflow information particular to this call site.

The implementation of type feedback in the SELF-93 optimizing compiler has been described elsewhere ([Höl94], [HU94a]). With type feedback, the SELF-93 compiler can inline more message sends and achieve better performance than previous compilers [HU94a]. For example, SELF-93 executes a suite of three medium-sized (400-1,100 lines) and six large (4,000-15,000 lines) programs 1.5 times faster than the SELF-91 compiler [HU94a]. For two medium-sized programs that are also available in Smalltalk, SELF-93 is about three times faster than ParcPlace Smalltalk.

In contrast to a type inference system (and to previous SELF systems), SELF-93 performs very little dataflow (or type-flow) analysis in an effort to keep the compiler small and fast. The compiler only performs trivial propagation of result and argument classes during inlining. For example, when inlining the send `foo: 1` the compiler will keep track of the fact that `foo`'s argument is the integer 1. On the other hand, when compiling the statements `j: i. i: i + j` the compiler will test j's class even if a simple dataflow analysis would reveal that j is equal to i.

2.4 Type inference

Concrete type inference or constraint-based analysis [PS91][APS93][PC94a], unlike type feedback, does not rely on executing the program. Given the program source, this global analysis will statically compute a type for every expression in the program. The types, as in type feedback, are sets of classes, but unlike the types obtained by running an instrumented program, the types computed by type inference are *safe* approximations:

- If the type {$\texttt{Class}_1$, $\texttt{Class}_2$, ..., $\texttt{Class}_\texttt{i}$} is inferred for some expression E, it is guaranteed that during *any* execution of the program, *every* time E is evaluated, the result is an object of $\texttt{Class}_1$, $\texttt{Class}_2$, ..., or $\texttt{Class}_\texttt{i}$.

The key idea in type inference, and one that sets it apart from traditional data flow analysis, is to compute control flow and data flow information simultaneously. This is necessary to analyze dynamically dispatched sends precisely because:

- to determine the methods a send may invoke, the possible classes (the type) of the receiver must be known, and
- to determine the type of a send, the methods it may invoke must be known.

Figure 2 shows a typical situation during type inference. A send, `obj->moveTo(0,0)`, is being analyzed. Previous inference has determined that the receiver expression, `obj`, may evaluate to a `Point` object, so the send is analyzed by connecting it to a *template* for the `Point::moveTo` method. A template is a representation of the control- and data flow within a method; if the method contains sends, these will yield connections to other templates. When a send is connected to a template, the types of the actual arguments, in this case simply {`Integer`}, are propagated into the corresponding formal arguments in the template. The type returned by the invoked method is determined by propagating the formal argument types through the template to its output. Finally, the result type of the template becomes the type inferred for the send connected to it.

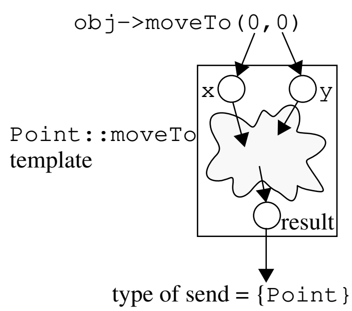

Figure 2. A send is analyzed by connecting it to a template for the invoked

As type inference proceeds, new classes of objects may be found to be possible receivers of previously analyzed sends. For example, the type inferencer may discover an assignment of a `Rectangle` object to the variable `obj`. Subsequently, it must be assumed that the send `obj->moveTo(0,0)` can invoke either `Point::moveTo` or `Rectangle::moveTo`. This situation is handled by connecting the send to templates for both of these methods and collecting the result types from each of them, as shown on Figure 3.

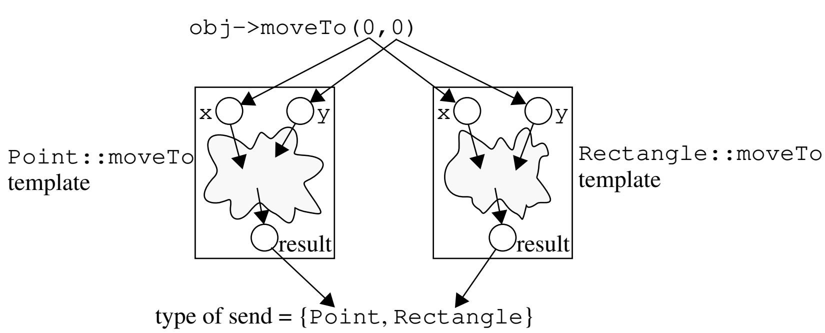

Figure 3. When a send may invoke several methods, it is connected to a template for each of them, and the send's type is obtained as the union of the templates' result types.

Type inference starts in a designated *main method* (equivalent to the `main()` function in a C program) and from there traces sends to other methods, creating templates for these as they are encountered, and recursively processing their sends. Eventually, when all possible method invocations have been analyzed, type inference is complete and a type is available for every expression in the program.

Polymorphism increases code reuse by allowing a piece of code to work on several kinds of objects. For example, a sort routine that can sort any vector of objects implementing "≤" is polymorphic. Polymorphic code is harder to analyze precisely because the different uses need to be kept distinct. To accomplish this, our type inference algorithm is *polyvariant*, i.e. may analyze each method more than once. Expressed in terms of templates, polyvariance means that several templates may be created for a single method, with different sends invoking the method being routed to different templates. Several polyvariance strategies, varying widely in precision and efficiency, have been proposed; see [Age94a] for an overview. The strategy used in this paper, the Cartesian product algorithm, computes the Cartesian product of the actual argument types and analyzes each combination separately. Details can be found in [Age95].

2.5 Summary and Comparison

Both type feedback and type inference operate with types that are sets of classes, and both systems produce types that are approximations. However, the approximations differ in nature. Type feedback *underestimates* the exact types, i.e., computes lower bounds—no matter how long a given expression is observed, the possibility that it may yield an instance of a new class next time remains. Type inference, on the other hand, *overestimates* the exact types, i.e., computes upper bounds. Type inference simply must approximate to remain computable. For example, finding the exact type of this conditional statement is hard (or at least was until recently):

```
if "Fermat's last theorem is true" then point else rectangle;
```

The lower bounds versus upper bounds distinction has important consequences for how the types can be applied during compilation. We discuss this issue in sections 3 and 4.

The other major difference between type feedback and type inference is that the former is *dynamic*, i.e., requires the program to be executed, whereas the latter is *static*. This difference also has consequences for the system as a whole, as will be discussed in section 4.

3. Quantitative evaluation

3.1 Implementation overview

To measure the usefulness of type inference for optimization, we combined and extended two existing systems: a type inferencer and application extractor written in SELF, and a SELF compiler written in C++. The interface between the two systems is a "snapshot" (image) containing a type-annotated benchmark program.

We first modified the application extractor, described in [Age94b], to output benchmark programs together with their type information. The normal functionality of the extractor is to identify a set of methods and objects that are sufficient to run a given application and write them out as SELF source code. The modified extractor additionally annotates each object with its "group ID," an integer representing the object's class. All objects with the same group ID are guaranteed to have the same implementation as far as the SELF virtual machine is concerned. Furthermore, each method is annotated with a "message ID," an integer serving as an index into an array of method templates. Each method template is a vector describing the inferred type for the method receiver, its result, and each of the expressions in the method. In the extracted format, a type is represented as a vector of group IDs. The extractor places all of these data structures at the end of the file containing the program source itself. Finally, the SELF virtual machine is invoked to convert the source file to a binary snapshot containing all the objects, methods, and type information. This snapshot is the only input to the type-inference based compiler.

We then modified the optimizing SELF-93 compiler to take advantage of this type information and to not use type feedback. Using the template array and several auxiliary arrays, the compiler searches for applicable method templates whenever it compiles a particular source method. In general, the type inferencer may generate more than one template per method, each of them for a particular combination of receiver or argument types [Age95]. Using its internal type information about the method's receiver and arguments, the compiler discards all templates that do not match (e.g., because they specify an incompatible receiver or argument type). Then, it merges the remaining templates by merging their entries and translates the resulting types into its internal type representation. While compiling the method, the compiler then uses this type information to inline messages. All other optimizations usually performed by the SELF compiler (such as customization, splitting, copy propagation, etc.) are performed as usual.

Whenever the compiler has a choice between using the information obtained from the type inferencer or its own information (obtained, for example, by simple local propagation or by optimizations such as constant-folding), the compiler chooses the more precise information. For example, the compiler might know that the result of the expression 3 + 4 is the constant 7 (and thus of type `{Integer}`) whereas type inference would give its type as `{Integer, BigInteger}`. Thus, the type inferencer benefits from the same local analysis that the standard system uses, so that the two can be fairly compared.†

The current SELF implementation requires that all methods be customized to their receiver [CUL89], and thus methods will be customized to a specific receiver even if the type inferencer does not require it. Similarly, the run-time system does not support multiple dispatch, so that the compiler may have to merge some method templates that differ only in their argument types.

To evaluate type feedback, we used an essentially unmodified version of the current SELF system which is based on the SELF-93 compiler [HU94a]. To enable an unbiased comparison, we made sure that the compiler used exactly the same optimization parameters and heuristics as the compiler using type inference.

3.2 Benchmarks and systems

To evaluate the relative performance of type inference and type feedback, we executed a suite of 23 benchmarks (see Table 1). The benchmark programs can be divided into three groups:

- “Tiny” is a set of very small integer benchmarks on which one would expect type inference to do particularly well since these programs do not use polymorphism. They are included for reference only.
- “Small” is a set of small benchmarks which primarily operate on integers and arrays and contain little polymorphism. These benchmarks are intended to represent the kernels of computationally intensive programs.
- “Large” is a set of application programs which were written by several different programmers and exhibit a variety of object-oriented programming styles. These programs most closely approximate typical SELF applications. One of them (`PrimMaker`) uses dynamic inheritance, i.e., objects that change their inheritance structure on the fly.

Because the programs in the Large suite are the most realistic, we will examine their behavior in detail throughout this paper while summarizing the programs in the other two sets. Full data on all benchmarks is given in the appendix.

To illustrate the various effects and trade-offs of type inference and type feedback, we measured several systems (see Table 2). The first two systems use type inference. TI is the standard configuration running unchanged source code with unchanged semantics; like Smalltalk, this system automatically converts “small” (30-bit) integers into arbitrary-precision integers if needed (except for RSA, none of the benchmarks actually use arbitrary-precision integers). Since the type inferencer does not perform any range analysis, this automatic conversion means that the result of adding two objects of type `{Integer}` is `{Integer, BigInteger}`. The second system, TI-int, prevents this conversion by treating integer overflow as a failure that halts the program, so that the result of adding two integers is always of type `{Integer}`. In all other aspects, TI-int is identical to TI. Finally, TF is the standard SELF compiler using type feedback and adaptive optimization. We do not distinguish between TF and TF-int because the two are virtually identical in performance.

Unless mentioned otherwise, all data in this paper are dynamic, i.e., take the relative execution frequencies of sends into account. To streamline the exposition, we usually give only summary charts in the main text, but detailed data can be found in the appendix. Box charts summarizing the large benchmarks can be found in the appendix on page 25.

† We believe that any optimizing compiler would use comparable or better local analysis, so that it would be unrealistic to compare a type inference system with purely global analysis to any other system.

	Name	Appl. size[a]	Total size[b]	Description
"Tiny"	AtAllPut	3	1,059	store 7 into all elements of 100,000-element vector
	SumTo	3	1,049	sums all integers between 1 and 10,000; repeated 100 times
	Recur	3	1,047	tiny recursive benchmark
	Tak	10	1,055	derived from Tak benchmark in the Gabriel Lisp benchmark suite
"Small"	Bubble	20	1,089	sort an array of 5,000 numbers with Bubblesort
	Detabify	21	1,071	replace tabs by blanks in a string of ASCII characters
	Intmm	30	1,092	40x40 integer matrix multiply
	Mergesort	50	1,169	sorts a 20,000-element array of integers using MergeSort
	Perm	25	1,082	heavily recursive permutation program
	Puzzle	170	1,309	solves a tile placement problem
	Queens	35	1,094	solves the eight-queens placement problem 50 times
	Quick	35	1,101	sort an array of 5,000 numbers with Quicksort
	Quick2	35	1,180	like quick, but written in an object-oriented style
	Sieve	25	1,053	computes prime numbers using the sieve of Eratosthenes (sieve size 8191)
	Towers	60	1,116	solves Towers of Hanoi problem for 14 disks
	Tree	25	1,108	sorts 5,000 random integers by inserting them into a sorted binary tree
"Large"	Deltablue	500	1,358	DeltaBlue constraint solver
	Diff	300	1,992	compares two files using the same algorithm as the Unix diff utility
	SParser	400	1,442	parser for the SELF-89 language
	PrimMaker	1,100	2,241	generates SELF and C glue stubs from a description of external C functions
	Richards	400	1,284	simulates a simple operating system
	RSA	300	1,541	public-key encryption and key computation (uses BigIntegers)
	CParser	7,000	10,984	parser for ANSI C; includes lexer, LALR(1) parser, and tree builder

Table 1. Benchmark programs

[a] Approx. lines of code, excluding code in standard classes such as integer, lists, etc. All line counts exclude blank lines.
[b] Lines of code that type inference shows to be part of the application, e.g., including methods in standard data types such as lists, arrays, etc. This size is *much* smaller than the full SELF environment, but may still contain some dead code. The line counts where obtained on versions of the programs that includes BigIntegers. Without BigIntegers, programs are consistently 450 lines shorter.

Name	Description
TI	SELF compiler modified to use type inference
TI-int	same as TI, but arbitrary-precision integer arithmetic is disabled
TF	SELF compiler using type feedback
unoptimized	SELF compiler with optimization turned off (no message inlining)

Table 2. Systems used in the study

3.3 Execution time

Speed is the ultimate goal of an optimizing compiler, and thus we start our analysis with bottom-line performance numbers; later sections will go into more details. Figure 4 shows the relative execution time of the benchmarks compiled with TF and the two TI systems. On average, TI-int is fastest, executing the large benchmarks a median of 15% faster than TF. TI is slower than TI-int, outperforming TF on only two of the seven large benchmarks. On the small integer benchmarks, TI's performance is very poor, two times slower than TI-int. Apparently, the compiler could not optimize these benchmarks well after predicting `{Integer, BigInteger}` receivers for the extremely frequent arithmetic operations.

The speedup of TI-int relative to TF should be taken with a grain of salt since TF could handle arbitrary-precision integers if they occurred whereas TI-int couldn't. However, integer arithmetic is not a dominant component of the large benchmarks, so the distortion of the results should be small.

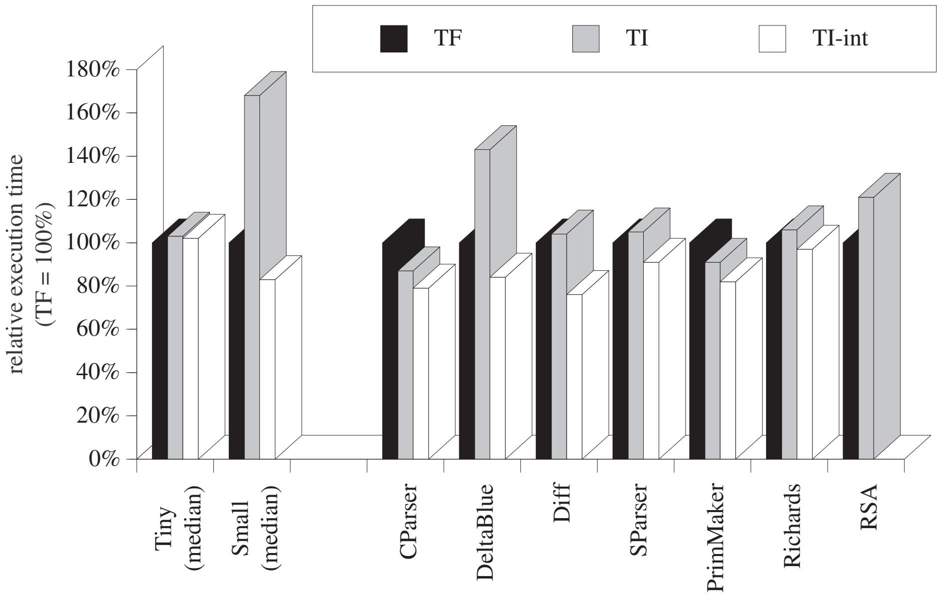

Figure 4. Relative execution time of benchmarks

TI-int (type inference without BigIntegers) usually is the fastest system. TI performs poorly on some benchmarks because the extra code to handle BigIntegers enlarges compiled methods and negatively affects other optimizations (see section 3.5 for an example).

As we will see later (in section 3.5), TI usually removes more dispatches than TF, and that is the main reason for TI's reduced execution time. SELF programs typically spend about 15% of their execution time in type tests implementing message dispatch [Höl94] which agrees well with the speedups measured here. Of course, other factors may also contribute to performance differences (e.g., instruction cache misses) but a detailed analysis is beyond the scope of this paper.

Although execution speed is important, it only summarizes the final outcome of many interacting processes. In the next sections we will examine several performance-related issues in more detail. Table 3 contains an overview of the detailed measurements presented in the remainder of this section.

Section	Data measured	Motivation	Main result
3.4	non-inlined sends	message inlining is important for performance	all systems inline equally well
3.5	number of dispatches[a]	ultimate goal of all optimizations is to eliminate dispatches (and the associated overhead)	TI-int outperforms all others; TI only marginally better than TF and sometimes worse (!)
3.6	degree of polymorphism	reflects improvements in precision of type information even if it doesn't lead to complete elimination of dispatch	same as above
3.7	estimated number of dispatches in other languages	extrapolate results to other languages where integers and booleans aren't objects (e.g., BETA, C++, Eiffel, Modula-3, Oberon)	no significant change; all systems optimize dispatches as well as in original SELF system

Table 3. Overview of detailed measurements and results

[a] "Dispatch" does not imply "call" in our terminology; see section 3.5.

3.4 Number of message sends

One of the main goals of optimizing compilers for pure object-oriented languages is to inline message sends. How well do type feedback and type inference perform in this respect? Figure 5 shows that both techniques inline a large fraction of message sends. For the smaller integer benchmarks, all configurations inline virtually all sends; usually, less than 1% of the original sends remain. In the large benchmarks, somewhat more non-inlined message sends remain, on average between 4% and 5% of the original sends. The data excludes non-dispatched calls† since they could be inlined if desired, but the results are very similar when all sends are included.

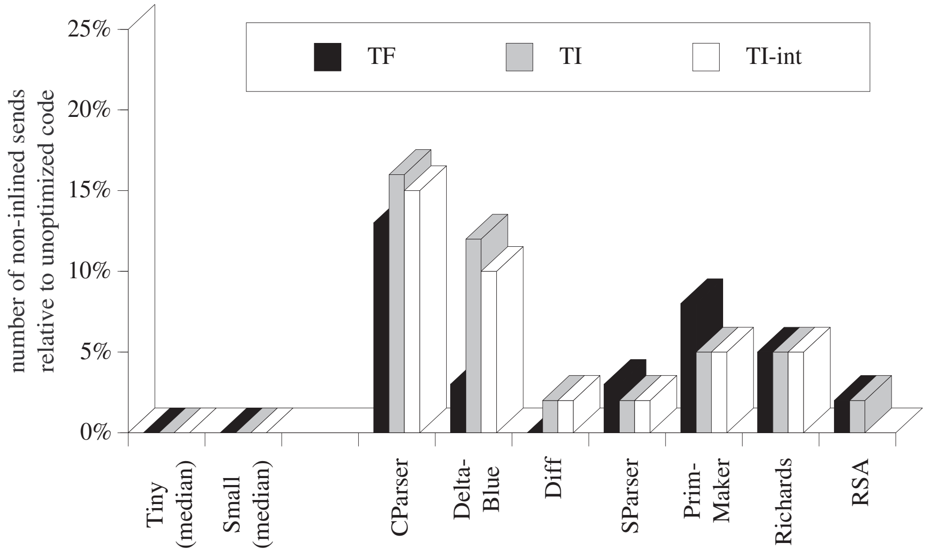

Figure 5. Number of non-inlined message sends relative to unoptimized SELF

This graphs shows that all systems are very successful at inlining message sends; often, less than 5% of the original calls remain. The graph exaggerates the differences between the systems somewhat since the absolute number of calls is often small. For example, the seemingly large relative difference in DeltaBlue represents an absolute difference of less that 90,000 calls (see appendix) and thus has very little effect on performance given the overall execution time.

A closer look at the large benchmarks reveals no striking differences between TF and TI for most benchmarks—both inline virtually the same proportion of message sends on almost all programs. The reason for this parity in inlining performance is simple: both systems have enough type information to inline virtually all sends. Thus, the remaining sends were not inlined because the compiler estimated that it was not worthwhile, e.g., because the method was considered too large. The number of remaining sends is therefore a function of the compiler's inlining policy (which was the same in all systems), and any variations are caused by factors unrelated to the type information per se.‡

3.5 Number of dispatches

A *dispatch* is the selection of the correct piece of code for a particular (receiver, message) pair. Even when a message send is inlined, it may still require a dispatch if more than one receiver class could occur. In other words, whether or not a send requires a dispatch is *independent* of whether it requires a call. Indeed, most dispatches select inlined methods, since the vast majority of message sends are inlined as shown above. By measuring the number of dispatches that a program performs, we can determine how successful the optimizer was in creating monomorphic call sites where dispatch can be avoided. We include both inlined and non-inlined dispatches because both introduce similar overhead.

The SELF system implements all dispatches via type test sequences which sequentially compare the receiver against predicted classes. For non-inlined sends, the type test is part of an inline cache or polymorphic inline cache [HCU91], and for inlined sends it surrounds the inlined code (see Section 2.3).

Type inference can determine the exact receiver class for many sends. Since such sends no longer require a dispatch, the overall number of dispatches is reduced.†† In contrast, type feedback requires a dispatch test even if it predicts a

† I.e., sends that do not require any dispatch. Examples include sends where type inference determined a single receiver class (but did not inline the send), or implicit-self sends (which require no dispatch because of customization [CUL89]).

‡ For example, the system using type feedback compiles methods in different order (since it adaptively optimizes the program [Höl94]) and thus generates a different set of methods.

single class, because the receiver type always includes the unknown class. Therefore, one would expect TI-compiled programs to execute strictly fewer dispatches than TF-compiled programs (as long as both systems use the same local analysis to propagate type information within a compiled method).

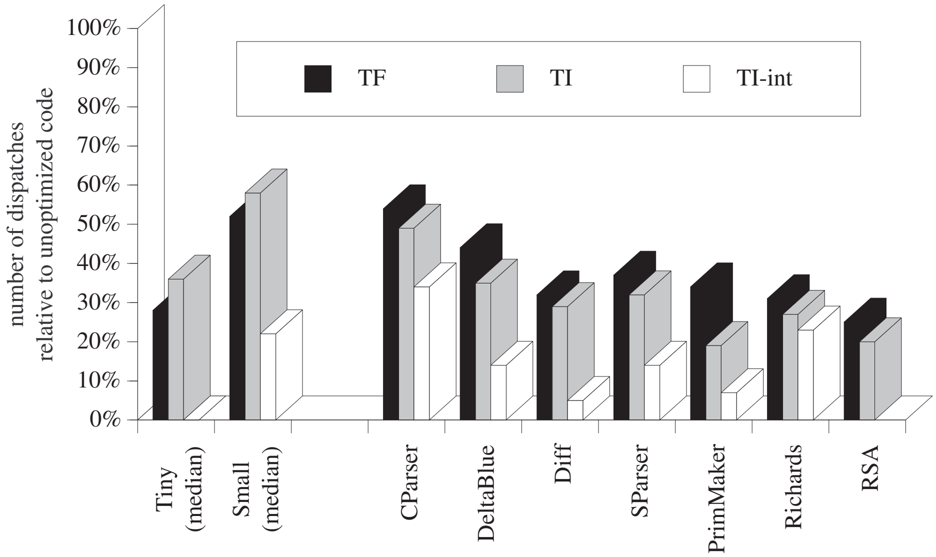

Figure 6. Type inference can eliminate more dispatches than type feedback
Note: "dispatch" does not imply "call" but also includes the dispatch of inlined sends (see text on page 9).

Figure 6 shows the number of remaining dispatches in optimized programs (relative to the dispatches performed in unoptimized programs). At first sight, it seems to confirm our expectations: on all benchmarks, TI-int performs fewer dispatches than TF, which already eliminates two thirds of all dispatches with its simple local analysis combined with customization and splitting. On average, TI-int executes 2.5 times fewer dispatches than TF. TI does not do as well but still consistently executes fewer dispatches than TF for the large benchmarks.

Unfortunately, TI's performance on the integer benchmarks squarely contradicts our expectations: on almost all integer programs, TI (i.e., type inference in the presence of arbitrary-precision BigIntegers) performs *more* dispatches than TF. On average, TI performs 8% more dispatches than TF, and `Bubble` even shows a disturbing 40% difference to TF (see the appendix for detailed numbers). What is going on?

The reason for the additional dispatch tests is subtle and requires a discussion of some details of the SELF-93 optimizing compiler. One of the optimizations it performs is *message splitting* [CU90]. Splitting avoids dispatches by copying parts of the control flow graph. In the code shown in the left half of Figure 7, the send of `area` requires a dispatch since its receiver could be either a circle or a square. Splitting duplicates (or "splits") the send and moves the copies into the two branches where it can be optimized (right side of Figure 7). To keep code expansion at a reasonable level, the current system only splits a send if the amount of unrelated code that needs to be copied ("other code" in Figure 7) is small.

Figure 8 shows a simplified code segment from the TI system. The code represents the expression "i < j", and type inference has determined the types of i and j to be {`Integer`, `BigInteger`}. After compiling the expression, the compiler encounters a send of `ifTrue:`[†] to the expression's result. The compiler could optimize this send by splitting the `ifTrue:` message into three copies (two for the integer case and one for the `BigInteger` case), but it elects not to split because it would have to duplicate too much code. Therefore, the send of `ifTrue:` needs a dispatch since its receiver type is {`true`, `false`}. In the corresponding TF program, however, the `BigInteger`

[††] Even when the receiver type contains multiple classes, it may still be possible to eliminate the dispatch if all receiver classes lead to the same method. Whether or not it is advantageous to exploit such an opportunity is a non-trivial question (for an extensive discussion see [DGC94]).

[†] In SELF, as in Smalltalk, `if` statements are conceptually implemented by message sends.

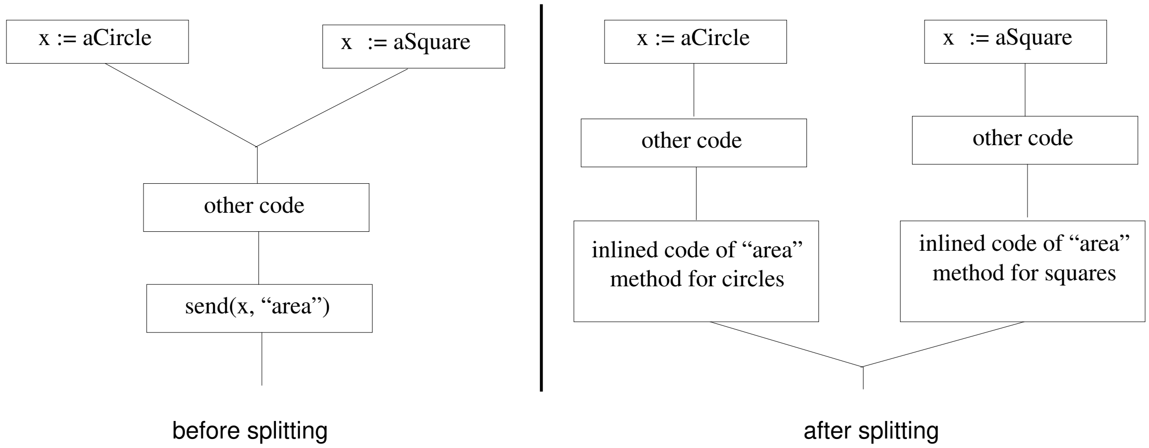

Figure 7. Splitting

In the code on the left, the send of `area` requires a dispatch since its receiver type is {`Circle`, `Square`}. Splitting copies the send (and any code between it and the control flow merge that created the union type) into the two branches before the merge. There, they can be inlined without a dispatch because the receiver class is known precisely.

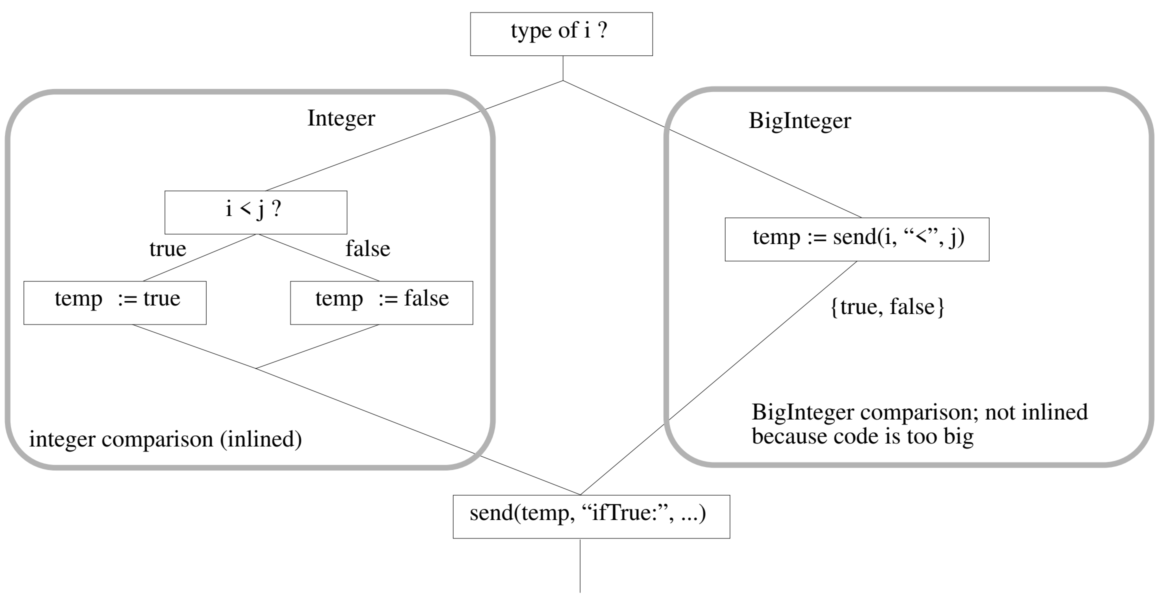

Figure 8. Intermediate code for the expression “i < j” in TI

TI compiles both Integer and `BigInteger` versions of the “less than” message since it has computed a receiver type of {`Integer`, `BigInteger`}. The result type is {`true`, `false`} (in SELF, the two boolean objects have distinct implementations, e.g., different methods for `ifTrue:`). Here, the comparison is followed by an `ifTrue:` send (as is often the case) which could be split as in Figure 7. However, the compiler decides not to split (thus leaving a dispatch for `ifTrue:`) since it would have to duplicate too much code. In TF, where the BigInteger part is replaced by a conditional trap, the compiler does split `ifTrue:`, creating a situation where TF performs fewer dispatches than TI.

branch does not exist (since only small integers occurred in the past), and the compiler does elect to split the `ifTrue:` send. Thus, TF eliminates a dispatch that was not eliminated in the TI system, and as a result TF performs fewer dispatches than TI (but not fewer than TI-int which also splits `ifTrue:`). Fortunately, at least the large (more realistic) programs behave as expected (Figure 6).

While this particular example is specific to SELF, the problem is more general and allows us to make two observations. First, this example shows that the value of type inference for optimization cannot be discussed in isolation from other compiler optimizations—in any compiler, optimizations may interact in unpredictable (or at least counter-intuitive) ways. Second, the example illustrates that the precision of the type information itself isn't necessarily a good predictor of performance. In the example, TI has more precise type information on i and j than TF ({`Integer`, `BigInteger`} vs. {`Integer`, `unknown`}), yet TF can produce better code since its information includes data on the relative frequencies of receiver classes.

3.6 Average degree of polymorphism

The previous section discussed how often optimized programs achieved the optimum of zero dispatch overhead per send. This section examines the "narrowness" of the receiver type information more generally by measuring the degree of polymorphism exhibited by dispatches. We characterize polymorphism by determining the *arity* of a dispatch, i.e., the number of possible receiver classes for that send. With type inference, arity is simply the cardinality of the receiver type set. For example, a send with a receiver type of {`Integer`, `BigInteger`} has an arity of two. With type feedback, arity is the number of predicted receiver classes for a particular send, plus one for the unknown class. For example, a send predicted for integers has an arity of two since its receiver type is {`Integer`, `unknown`}. A perfectly monomorphic program will thus have an arity of 1.†

The arity is averaged over all sends logically performed by the source program, regardless of how they are implemented at run time (i.e., whether they are inlined or not). A send without a dispatch test is counted with arity 1. Average arity is related to the compiler's success in removing dispatches entirely—programs with many eliminated dispatches will have a low average arity. However, unlike the black-and-white measure of the previous section (where a send is either eliminated or not, with nothing in-between), average arity reveals more shades of gray. A system that reduces the average arity to 1.2 is arguably better than another system with an arity of 1.4 since the first system's type information is narrower. Even if both systems currently eliminate the same number of dispatches, the first system is likely to eliminate more dispatches if further optimizations were introduced.

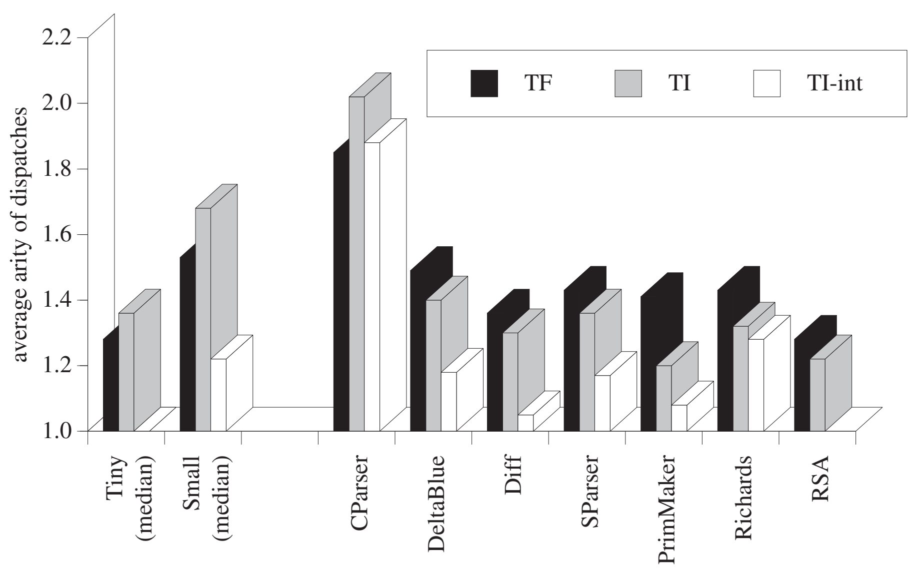

Figure 9. Arity of type tests

This graph shows the average arity (i.e., degree of polymorphism) for all message sends, whether they have been inlined or not. (A send requiring no dispatch has an arity of one.)

Figure 9 shows the average arity of message sends in the benchmark programs.‡ Since 1 is the lower bound on arity, the y axis starts at 1 instead of 0. TI-int's arity is about two times closer to the ideal (1.0), but TI's arity is only slightly

† Since boolean expressions have the type {`true`, `false`} in SELF (rather than, say, {`Boolean`}), conditional tests may involve dispatches (although splitting usually eliminates many of these). Consequently, very few SELF programs are perfectly monomorphic.

lower than that of TF (the box chart on page 25 illustrates this relationship well). In other words, type inference significantly reduces the degree of run-time polymorphism, but only if arbitrary-precision integer arithmetic is disabled. The results roughly mirror those of Figure 6 (remaining dispatches), where TI-int removed twice as many dispatches as TF.

An average arity of 1.2 for the integer programs may seem very high, indicating that many sends required a dispatch even in the optimized programs. SELF programs *are* unusually polymorphic since many common idioms involve polymorphism. Four cases are particularly important: integer arithmetic may overflow into BigIntegers, booleans are polymorphic (i.e., `true` and `false` have two distinct types), `nil` is an object rather than a special value (so that a reference containing either `nil` or a `Point` is polymorphic), and strings come in three varieties (canonicalized strings, mutable strings, and byte vectors). Thus, SELF programs probably stress type inferencers more than programs written in other languages. (`CParser`'s arity is especially high because it builds a parse tree with 200 different classes of nodes, one per non-terminal in the C grammar, using 300 different classes of action objects, one per production in the C grammar.)

3.7 Extrapolating to other object-oriented languages

All data presented so far is specific to SELF, in which even integers are objects. How relevant is this data to other object-oriented languages such as C++, Modula-3, or Oberon? An accurate answer, of course, can only be obtained by reimplementing type inferencer, compiler, and benchmarks in those other languages. Since this task was beyond our means, we instead used the SELF system to shed some light on this question by excluding dispatches that wouldn't occur in other languages, namely, dispatches on integers, floating-point numbers, blocks (closures), booleans, and `nil`. By excluding these dispatches, we simulate a hypothetical SELF-like language (which we will call SELF++) where the basic data types are non-objects and where operations on them are non-dispatched. Consequently, it would be impossible in SELF++ to write a polymorphic method that accepted either an integer or a user-defined `BigInteger` object as the argument. While we do not claim that SELF++ bears much resemblance to any real object-oriented language, it provides at least an indicator of how much the data presented so far might be biased by our use of SELF as the experimental vehicle. To clarify the discussion, we will call dispatches involving primitive objects (integers, floating-point numbers, booleans, and nil) "SELF" dispatches and all others "SELF++" dispatches.

Figure 10 shows that SELF's object model is responsible for a significant fraction of all dispatches. Typically, about half of all dispatches are SELF dispatches in the large benchmarks, and virtually all dispatches are SELF dispatches in the small integer benchmarks. In other words, unless nil, true/false, etc. are considered objects, the small integer benchmarks are completely monomorphic, procedural programs. The larger programs show a clear difference between unoptimized programs (which have a small fraction of SELF++ dispatches) to TF and TI. This difference is a result of inlining and splitting: in both TF and TI, most control structures are inlined, eliminating many dispatches to block objects that exist in unoptimized code. Furthermore, splitting eliminates many dispatches on boolean results (e.g., in `if` statements). Thus, unoptimized code has many "trivial" dispatches in SELF, and since none of them would occur in SELF++, dispatches in unoptimized code are reduced much more when going from SELF to SELF++ than in TF or TI where many of those trivial dispatches are optimized away. While Figure 10 also hints at a difference between TF and TI (with TI having a higher fraction of SELF++ dispatches), a closer look at the data (or the box chart on page 25) reveals that while TI's variance is much higher, it's mean and median aren't that different from TF. Based on this observation, we do not believe that the data indicates a significant difference between TF and TI in this respect.

How well would TF and TI perform in the hypothetical SELF++ system? Figure 11 shows optimization effects for the SELF++ system that are quite different from those seen for the SELF system. One measure is the fraction of dispatches eliminated relative to unoptimized code. Both TF and TI are less effective in reducing the number of dispatches executed than they were in SELF. TF executes a median 61% of the dispatches executed by unoptimized programs, up

‡ The data for unoptimized programs is not shown because it is too dependent on particular implementation choices made in the SELF system. Each block literal (closure) has a different class in SELF, so that control structures like `ifTrue:` are highly polymorphic in their arguments. This effect inflates the arity relative to a system where all n-argument blocks are instances of class NArgumentBlock (as is the case in many Smalltalk systems). The unoptimized programs actually implement control structures like `ifTrue:` as message sends, artificially inflating their degree of polymorphism. In the optimizing systems most control structures are inlined, and thus the effect of SELF-style blocks is quite small.

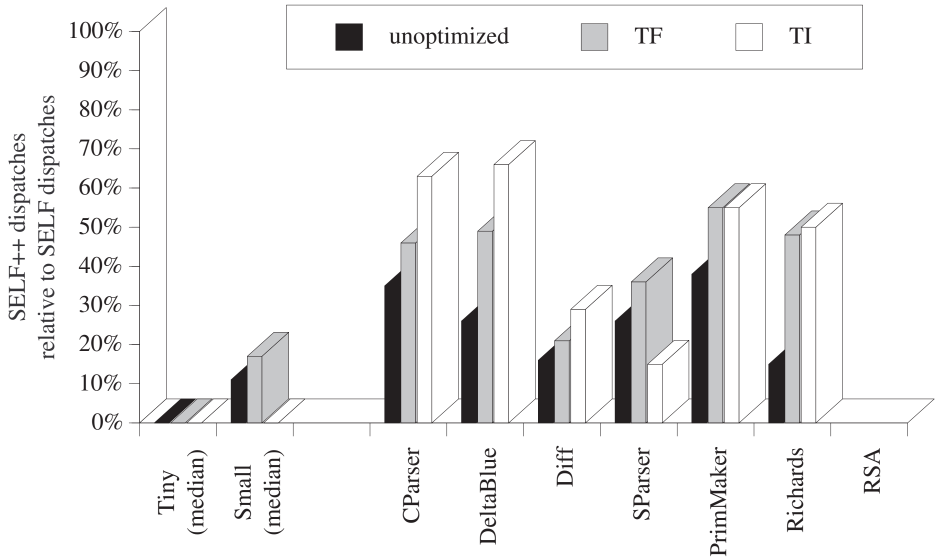

Figure 10. Dispatches in SELF++ relative to SELF

This graph shows the "SELF++" dispatches of programs, i.e., those that are not caused by SELF's pure language model that treats even "primitive" data (integers, floats, booleans, and nil) as objects. For example, about 75% of the dispatches in unoptimized Tiny programs involve primitive data, so that only about 25% of the dispatches would remain in an unoptimized program written in a hypothetical SELF++ language that treated primitive data as non-objects. Note that each bar is relative to its system: for example, a value of 30% for both TF and TI does *not* mean that TF and TI would perform the same absolute number of dispatches—it means that in both systems the number of dispatches would be reduced to 30% relative to the SELF system. The RSA benchmark uses mixed-mode arithmetic (adding integers to BigIntegers) and thus wouldn't run in SELF++.

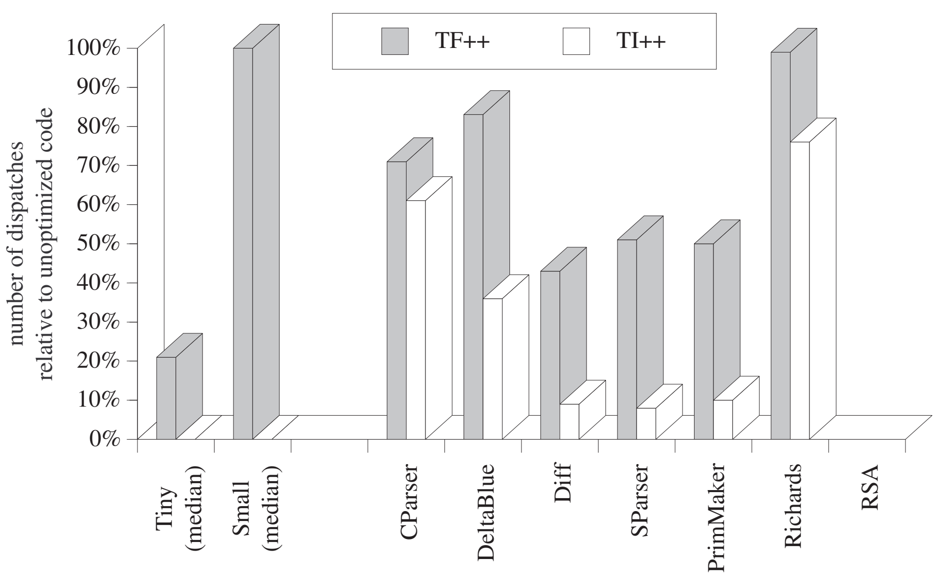

Figure 11. Remaining run-time dispatches in SELF++

This figure is equivalent to Figure 6, except that all dispatches involving primitive objects integers are excluded. (As in all other places, "dispatch" does not imply "call"—see section 3.5 on page 9.)

from 34% in SELF, an increase of a factor of 1.8. Similarly, TI only reduces dispatches to 23% of the unoptimized dispatches instead of 14% in SELF, an increase of a factor of 1.6. Apparently, dispatches to integers, boolean, and the like are easier to eliminate than dispatches to other objects. While TF's performance is fairly uniform, TI displays bimodal performance characteristics: it does very well on some programs (Diff, SParser, and PrimMaker) but considerably worse on others (CParser and Richards).

These results are only a rough extrapolation based on particular implementations of TF and TI and on a particular programming style. Based on the above data and our intuition, we predict that type feedback and type inference for languages like C++, Modula-3, or Oberon behave as follows:

- Type feedback will remove relatively few dispatches but will inline as many virtual calls as desired.
- Concrete type inference will be spectacularly successful in some programs but less so in others. In general, it will eliminate more dispatches (but inline about the same number of calls) than type feedback.
- If the type of frequently-executed expression is de facto monomorphic but theoretically polymorphic (as in SELF's integer-BigInteger arithmetic or in a statement like `if unlikely_condition then return errorObj else return obj`), type inference may not perform well without information on receiver class frequency.

Overall, we believe that type feedback and type inference behave similarly in other languages than they do in SELF, and that the results presented here can be used as a starting point for similar investigations of other object-oriented languages. Of course, measurements of implementations of the two techniques for other object-oriented languages are needed to substantiate this belief.

4. Discussion

While raw execution speed is often an important consideration, and an easy one to quantify, other factors will also influence the choice between type feedback or type inference. Indeed, since our measurements indicate that type feedback and type inference can deliver comparable and high performance, these other factors will likely decide the outcome. In this section we discuss the most important advantages and disadvantages of both kinds of systems in a broader context. Table 4 outlines the main characteristics that will be discussed in more detail below.

Characteristic	Type Feedback	Type Inference
Responsiveness	incremental, sub-second pauses; suitable for interactive systems	"batch style", multi-second to multi-minute pauses, not (yet) suitable for interactive systems
Performance	may perform poorly if statically compiled with non-representative profile data	may perform poorly without dynamic information
Application delivery	only need to generate compiled code for cases that actually occur	compiled code can cover all cases
	needs compiler or interpreter at runtime	can generate self-contained executable
	needs source code (or equivalent) at run time	no need to keep around source code
Generality	handles entire language; supports extensible systems	cannot handle entire language; may not scale to very large programs; doesn't support extensible systems
Implementation	can use same compiler for both development and delivery	needs two separate compilers (one for development, one for delivery)

Table 4. Main characteristics of type inference and type feedback

Compilation time—providing high responsiveness. When type feedback was developed for SELF, a major goal was to bless the programmer with high performance *and* the absence of compilation pauses. The goal was met largely due to the incremental nature of type feedback: type information is computed gradually and methods can be compiled one at a time (after inline expansion, of course) [HU94b]. On the other hand, the style of type inference used in this work is fundamentally a global analysis: the unit analyzed is an entire program. Obviously, inserting a global analysis into the edit-compile-run loop (or the edit-continue loop of the SELF system) will not go unnoticed by the programmer.

The SELF type inference system supports incremental recomputation of the types when a previously analyzed program is modified locally [Age95]. While an incremental re-analysis is often an order of magnitude faster than a full analysis, it is still slower than the typical sub-second compile pauses of type feedback. Unfortunately, our experimental type-inference-based compiler does not carry the incrementality through (as described in section 3.1), so we have not been able to quantify how well an incremental type-inference-based system may support an interactive environment. For these reasons, we consider type feedback the safer bet if interactiveness is a top priority.

Finally, while the type inference system is quite fast (using at most a few minutes of CPU time for the largest programs we measured), it needs considerable amounts of memory during extraction. For example, the largest programs measured here consumed about 80 Mbytes of memory during the type inference and extraction steps (out of these 80 Mbytes, the standard SELF system in which the inferencer runs accounts for approximately 32 Mbytes).

Consistent performance. Both systems may occasionally display poor performance. If type feedback is implemented statically (i.e., without run-time compilation), the quality of the generated code will depend on the quality of the training runs that supply the type feedback information, i.e., how closely these runs represent typical usage. Similarly, the static nature of type inference may lead to poor performance if very few of the statically predicted cases actually occur at run time. `BigInteger` arithmetic is a prime example of this problem: most arithmetic never overflows, but the type inferencer cannot prove this. For the small integer programs in our benchmarks, the missing dynamic information led to a slowdown of a factor of two over TI-int (recall Figure 4 on page 8). For consistently good performance, it may therefore be necessary to combine type inference with profile data in order to recognize such cases. On the other hand, arbitrary-precision integer arithmetic represents something of a worst-case scenario for type inference since integer arithmetic is a very frequent operation (e.g., every `for` loop includes it). In general, missing frequency information may therefore have a smaller impact than shown in TI (this is the reason why we included TI-int in the study).

Application delivery. Type inference gives conservative and sound estimates of the types, and thus accounts for all cases that may possibly occur during any execution of the program. This makes it possible to compile the entire program and ship an application as a stand-alone executable. In contrast, type feedback can at any given time "only" deliver type information that covers cases that have occurred hitherto; new cases may occur at any time in the future. At run time, an application must therefore have access to a compiler or interpreter to handle residual cases which were not encountered during the type feedback training runs.

If the type profile obtained during type feedback training runs accurately reflects typical application use, unpredicted cases will occur only rarely and can probably be handled efficiently by an interpreter. However, if a user stresses an application in unforeseen ways and heavily exercises code that was not optimized for that particular case, the resulting performance could be poor. For example, an application part optimized for graphical objects with integer coordinates would have to be interpreted if the user's objects contained floating-point coordinates. With an adaptively reoptimizing compiler like the SELF-93 compiler, the delivery runtime system could efficiently handle such cases, but such a compiler would of course occupy more space than a simple interpreter.

Application size. While we could not quantify how much larger the compiled code generated by a type inference based system will be, it is likely to be bigger than in a type feedback based system, With type inference, the code is based on a conservative estimate of program flow which is guaranteed to include all possible execution paths but which may also include paths that cannot be executed (after all, type inference can only approximate the true program behavior). Type feedback based systems, in contrast, contain code only for paths that were actually executed, i.e., for only a subset of all possible program executions. (Programs do include additional checks and trap instructions to cover unexpected cases [Höl94], but this code contributes little to the overall code size.) To cover all other executions, a type feedback system needs to include an interpreter or compiler at an additional space cost of as little as 20 Kbytes for a Smalltalk byte-code interpreter [Kra83] or as much as 800 Kbytes for an optimizing compiler [Höl94]. For smaller applications, the fixed cost of the interpreter or compiler will tilt the scales toward type inference, whereas larger applications will tend to favor type feedback. Since so many different aspects influence space usage, we are not convinced that either system is inherently more space-efficient.

Of course, type inference can significantly reduce program size through its use in program extraction [Age94b]. However, program extraction is performed in a separate step and can easily be used for application delivery with a type feedback based system (in fact, this is what our experimental implementation does).

Source code protection. Since an executable generated with the help of type inference is complete, it need not include the application's source code or any source code equivalent such as byte codes. Since optimized code usually represents a fairly thorough obfuscation of the original source code, an application is well protected against reverse engineering or source code theft. A type feedback system (or any other system based on dynamic compilation [DS84]) needs a source code equivalent to execute new cases as they occur. Since typical byte codes are fairly high-level, an approximation of the original source code can be generated easily, and thus source code isn't as well protected.

Generality. Although type inference systems for object-oriented languages have made much progress in recent years, they are still unable to effectively handle certain language constructs (and indeed there is little hope they ever will). Most of these limitations relate to reflective operations such as constructing new classes at runtime or sending messages whose names are computed at runtime ("performs"). For programs using these features extensively, type inference may not be effective. Furthermore, we currently have no experience with type inference on very large programs (say, a few hundred thousand lines of code long), and thus it is unclear how well type inference will scale to such programs.

Static analysis techniques like type inference require knowledge of the complete program, which can be a serious problem in today's extensible application environments. For one, many programs are dynamically linked, so that the exact implementation of the dynamic link library is not available until run time (its interface is known but of limited help to concrete type inference). Furthermore, many programs are extensible, i.e., allow the user to dynamically link in application extensions provided by third parties. Adapting concrete type inference to work in such an environment remains a major technical challenge.

5. Related work

Type inference for dynamically-typed object-oriented languages has been an active field since the early eighties. Suzuki pioneered the field in the Smalltalk community [Suz81]. He also used sets of classes as his basic types and had the same goal of eliminating dynamic dispatch to improve execution efficiency. The type inference system was an adaptation of Hindley/Milner style type inference, extended with support for assignable variables, union types, and—most importantly—an iterative analysis technique to handle dynamically dispatched sends. Unfortunately, Suzuki was unable to fully implement his system within the limited heap space of Smalltalk-76 and only managed to test a simplified version of the inference algorithm on some of the number classes.

The Typed Smalltalk project [JGZ88] incorporated a type inferencer for Smalltalk which relied on explicit type declarations inserted by the programmer; these types were specified as sets of classes. (A second, more abstract kind of type was provided for type checking but was not used by the compiler). Unfortunately, the system was not completed and the only published performance data concerned very small integer programs similar to our Tiny benchmark set.

More recently, Palsberg and Schwartzbach described a constraint-based analysis system for a Smalltalk-like toy language [PS91], which was improved and implemented in cooperation with Oxhøj [OPS92]. Palsberg and Schwartzbach suggested the use of their types for optimization but did not implement such a system. The type inferencer used here is a further development of the ideas originally proposed by Palsberg and Schwartzbach.

Plevyak and Chien independently improved Palsberg and Schwartzbach's algorithm for Concurrent Aggregates (CA), a concurrent, single-inheritance, dynamically-typed object-oriented language [PC94a]. Their system employs an iterative analysis which in each iteration uses the types from the previous iteration to guide the analysis.The inference algorithm can clone classes during analysis to obtain high precision on code with *data polymorphism*. (Our algorithm is less precise in this regard. It only clones classes based on observing initial values of their instances). Plevyak and Chien's type inferencer has been used in the CA compiler to eliminate dispatches in the (non-object-oriented) Livermore Loops [PZC95] and in object-oriented CA programs [PC94b][PC94c]. In the latter two studies, the compiler was able to completely eliminate dynamic dispatch in six of the nine benchmark programs. Because of language and application differences (in particular, different treatment of booleans, integers, and nil), it is hard to compare these results to the results presented here or to estimate how well type feedback would perform in the CA system.

Recently, Pande and Ryder have made progress in extending data-flow techniques to C++ [PR94]. The main goal of their work is to extend dataflow analysis techniques to solve static analysis problems for C++. They state—and we agree—that the first problem to solve is that of computing concrete type information, since without it, many other analyses fall apart due to the lack of precise control-flow information for virtual calls (i.e., dynamically dispatched sends). Similar to Palsberg and Schwartzbach, Plevyak and Chien, and the present system, the analysis starts with a control flow graph without virtual invocations which is subsequently updated whenever new virtual methods become invocable during type propagation. For C++, type inference is both harder and simpler than, say, for SELF. It is harder because C++ is not type- and pointer-safe and has many language constructs that are difficult to handle. On the other hand, C++ does not have user-defined control structures, closures, or dynamic inheritance. Preliminary results indicate that their technique can eliminate a significant fraction of dispatches in small C++ programs containing a few dozen call sites [PR94].

Traditional data-flow analysis techniques cannot be straightforwardly applied to object-oriented languages because the existence of a control-flow graph is assumed prior to analysis. The strength of the analyses mentioned above ([PS91], [PC94a], [PR94], and [Age95]) is that they acknowledge this problem up-front and solve a combined control-flow and data-flow problem. Vitek et al. demonstrated that this can also be achieved in a data-flow framework [VHU92]: for each program point, they computed an abstract object graph, a sufficiently detailed approximation of the full program heap to allow simulation of lookups. However, no data was given about the effectiveness of this approach.

The SELF-91 compiler used iterative type analysis to eliminate dispatches [CU90]. Like the SELF-93 compiler used here, its analysis was local, i.e., restricted to a single method and the methods already inlined into it. However, its analysis was significantly more powerful than the simple analysis used in SELF-93.

SELF compilers were the first ones to use type feedback for object-oriented programs [HCU91], [HU94a]. Other systems have used some form of runtime type information for optimization. For example, Mitchell's system [Mit70] specialized arithmetic operations to the run-time types of the operands (similar to SELF-89's customization [CUL89]). Similarly, several APL compilers created specialized code for certain expressions (e.g. [Joh79], [Dyk77], [GW78]). The HP APL compiler [Dyk77] specialized compiled code according to the specific operand types (number of dimensions, size of each dimension, element type, etc.). This so-called "hard" code could execute much more efficiently than more general versions since the cost of an APL operator varies greatly depending on the actual argument types. If the code was invoked with incompatible types, a new version with less restrictive assumptions was generated (so-called "soft" code). Garrett et al. [Gar94] describe a compiler for the object-oriented language Cecil that uses type feedback in combination with static compilation to eliminate dynamically-dispatched calls.

Dean et al. [DGC94] eliminate dynamic dispatch with a simple class hierarchy analysis that detects situations where a method has no overriding definition in any subclass of the sending method holder. Their study is the only other study we are presently aware of that compares a static analysis technique (class hierarchy analysis) to a dynamic technique (type feedback). For a large Cecil program, class hierarchy analysis reduced the number of dispatches by 20%; this smaller reduction compared to the reductions reported here can be explained by the more powerful analysis performed by TI. In contrast to the relatively small absolute performance differences in our study, Dean et al. report more significant performance differences on the four Cecil programs measured: when used in isolation, type feedback was a median 1.3 times faster than class hierarchy analysis. Combining type feedback with class hierarchy analysis resulted in an additional median speedup of 1.4 over type feedback alone. However, as noted by the authors, the Cecil compiler underlying the study is still under development, so that performance differences may be exaggerated by missing optimizations such as splitting and selective customization.

6. Conclusions

Both type feedback and concrete type inference are valuable techniques for optimizing object-oriented programs. We have presented a detailed comparison of these two techniques. To our knowledge, this is the first in-depth comparison of a static and a dynamic optimization technique for object-oriented programs.[†] Our comparison is particularly interesting for two reasons:

[†] The only exception being section 3.3 of [DGC94] which contains some quantitative data of a comparison of type feedback and class hierarchy analysis.

- First, it is *direct*, because we eliminated other effects by using two identical compilers except for their source of receiver class information.
- Second, it is *realistic*, since the compared systems are high-quality implementations. Both are based on the SELF-93 system which has been shown to provide excellent performance compared to a commercial Smalltalk implementation [HU94a].
- Finally, although the quantitative results are of course specific to SELF, they may nevertheless be interesting to implementors of other languages. An experiment with SELF++, a hypothetical SELF-like language treating integers and other primitive data types as non-objects (as do, e.g., Beta, C++, Eiffel, etc.), indicated that the relative effectiveness of type feedback and type inference in such a system could be quite similar.

The specific quantitative results of this study include the following:

- Both techniques can deliver high performance and typically inline more than 95% of all sends in our suite of 23 SELF benchmarks. With the exception of arbitrary-precision arithmetic, the median performance of all systems differed by only 15%.
- Information about the run-time frequency of receiver classes is important for performance, as was demonstrated most clearly by SELF's arbitrary-precision integers where type inference could not rule out the possibility of arithmetic overflows whereas type feedback could quantify them as infrequent. Consequently, small integer-intensive programs compiled with type inference (TI) ran 70% slower than with type feedback (TF). It should be noted, though, that arbitrary-precision integer arithmetic represents a worst-case scenario for type inference, since `BigInteger` objects occur everywhere arithmetic is performed (e.g., loop counters). For larger programs less dominated by arithmetic, this shortcoming of type inference had a smaller overall effect.
- Without arbitrary-precision integer arithmetic, TI-int performed 2.5 times fewer dispatches than TF and improved performance by a median 15%. With arbitrary-precision integer arithmetic, TI performed 1.3 times fewer dispatches than TF on the large benchmarks but on some integer benchmarks performed *more* dispatches than TF.

Both techniques have their strengths and weaknesses. Type inference can reduce the number of dispatches executed (but may also increase them—see section 3.5), works statically (without any dependence on run-time information), and can produce complete executables so that no compiler or interpreter need be present at run time. On the downside, it may perform poorly in some situations (see above), is not yet suitable for interactive use, and does not support extensible programs. Type feedback is well suited to interactive systems, provides consistently high performance, and scales well to large and extensible systems. On the other hand, it usually executes more dispatches, depends on run-time information (and representative profiles, if using static feedback), and introduces run-time overhead for (re-)compilation or interpretation if using dynamic feedback.

An interesting direction for future work is to combine type feedback and type inference. The strengths and weaknesses of the two techniques are largely complementary, and thus either system might be able take advantage of the strengths of the other. For example, it might be possible to build a system where type feedback is in control but uses type inference to extend the analysis from the current local scope to a larger scope (i.e., beyond inlining), or a system where type inference is in control but defers analyzing certain cases until type feedback has revealed that they may occur. Finally (and most simply), a type inference based compiler could use dynamic profile information to select the cases to be inlined, rather than treating all cases as equally likely.

Acknowledgments. We would like to thank Sun Microsystems Laboratories for generously supporting the first author and for providing equipment that greatly facilitated this work. Furthermore, we are indebted to John Plevyak, Klaus Schauser, and Mario Wolczko for their comments on earlier versions of this report.

References

[Age94a] Ole Agesen. Constraint-Based Type Inference and Parametric Polymorphism. In *SAS'94, First International Static Analysis Symposium*, Namur, Belgium, September 1994. Springer-Verlag (LNCS 864).

[Age94b] Ole Agesen and David Ungar. Sifting Out the Gold: Delivering Compact Applications from an Object-Oriented Exploratory Programming Environment. In *OOPSLA'94, Object-Oriented Programming Systems, Languages and Applications,* Portland, OR, October 1994.

[Age95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. To be published at ECOOP'95.

[APS93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance. In *ECOOP '93 Conference Proceedings*, p. 247-267. Kaiserslautern, Germany, July 1993.

[CGZ94] Brad Calder, Dirk Grunwald, and Benjamin Zorn. *Quantifying Behavioral Differences Between C and C++ Programs*. Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.

[CUL89] 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*, p. 49-70, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October 1989.

[CU90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, p. 150-164, White Plains, NY, June 1990. Published as *SIGPLAN Notices 25(6)*, June 1990.

[DGC94] Jeffrey Dean, David Grove, and Craig Chambers. *Optimization of object-oriented programs using static class hierarchy analysis*. Technical Report CSE TR-94-12-01, University of Washington, December 1994.

[DS84] L. Peter Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. *Proceedings of the 11th Symposium on the Principles of Programming Languages*, Salt Lake City, UT, 1984.

[Dyk77] Eric J. Van Dyke. A dynamic incremental compiler for an interpretative language. *HP Journal*, p. 17-24, July 1977.

[Gar94] Charles D. Garrett, Jeffrey Dean, David Grove, and Craig Chambers. *Measurement and Application of Dynamic Receiver Class Distributions*. Technical Report CSE-TR-94-03-05, University of Washington, February 1994.

[GW78] Leo J. Guibas and Douglas K. Wyatt. Compilation and Delayed Evaluation in APL. In *Fifth Annual ACM Symposium on Principles of Programming Languages*, p. 1-8, 1978.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In *ECOOP'91 Conference Proceedings*, Geneva, 1991. Published as *Springer Verlag Lecture Notes in Computer Science 512*, Springer Verlag, Berlin, 1991.

[Höl94] Urs Hölzle. Adaptive Optimization in SELF: *Reconciling High Performance with Exploratory Programming*. Ph.D. Thesis, Department of Computer Science, Stanford University, 1994. (Available via ftp/http from self.stanford.edu.)

[HU94a] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, p. 326-336. Published as *SIGPLAN Notices 29(6)*, June 1994.

[HU94b] Urs Hölzle and David Ungar. A Third-Generation SELF Implementation: Reconciling Responsiveness with Performance. *OOPSLA '94 Conference Proceedings*, p. 229-243, Portland, OR, October 1994. Published as *SIGPLAN Notices 29(10)*, October 1994.

[JGZ88] Ralph Johnson, Justin Graver, and Lawrence Zurawski. TS: An optimizing compiler for Smalltalk. In OOPSLA '88 Conference Proceedings, San Diego, CA, October 1988.

[Joh79] Ronald L. Johnston. The Dynamic Incremental Compiler of APL\3000. In *Proceedings of the APL '79 Conference*. Published as *APL Quote Quad 9(4)*, p. 82-87, 1979.

[Kra83] Glenn Krasner, ed., *Smalltalk-80: Bits of History and Words of Advice.* Addison-Wesley, Reading, MA, 1983.

[Mit70] J. G. Mitchell, *Design and Construction of Flexible and Efficient Interactive Programming Systems*. Ph.D. Thesis, Carnegie-Mellon University, 1970.

[OPS92] Nicholas Oxhøj, Jens Palsberg and Michael I. Schwartzbach. Making Type Inference Practical. In *ECOOP'92 Conference Proceedings*, Utrecht, The Netherlands, 1991. Published as *Springer Verlag Lecture Notes in Computer Science 615*, Springer Verlag, Berlin, 1991.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Phoenix, Arizona, October 1991.

[PR94] Hemant D. Pande and Barbara G. Ryder. *Static Type Determination and Aliasing for C++*. Technical Report LCSR-TR-236, Rutgers University, December 1994.

[PC94a] John B. Plevyak and Andrew A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *OOPSLA'94, Object-Oriented Programming Systems, Languages and Applications,* Portland, OR, October 1994.

[PC94b] John B. Plevyak and Andrew A. Chien. Precise Concrete Type Inference and its Use in Program Optimization. Unpublished report, October 1994.

[PC94c] John B. Plevyak and Andrew A. Chien. Efficient cloning to eliminate dynamic dispatch in object-oriented languages. Unpublished report, December 1994.

[PZC95] John B. Plevyak, Xingbin Zhang, and Andrew A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the 22nd Symposium on Principles of Programming Languages*, San Francisco, CA, January 1995.

[Suz81] Norihisa Suzuki. Inferring Types in Smalltalk. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages,* p. 227-241, 1981.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, p. 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.

[VHU92] Jan Vitek, R. Nigel Horspool and James S. Uhl. Compile-time Analysis of Object-Oriented Programs. In *Proceedings of CC'92, 4'th International Conference on Compiler Construction*, Paderborn, Germany, pages 236-250. Springer-Verlag (LNCS 641), 1992.

Appendix A. Detailed Data

In the tables below, “calls” is the total number of calls (dispatched or not), “static calls” is the number of non-dispatched calls (a subset of “calls”), “dispatches” is the total number of dispatches (whether inlined or not, see section 3.5 on page 9), “avg. arity” is the dynamic average arity of all sends, and “time” is the CPU time (in seconds) on a 40MHz SPARCstation-10 workstation (most benchmarks were repeated 25 times to get more accurate timings).

Benchmark	calls	static calls	dispatches	avg. arity	time
AtAllPut	1,600,054	1,000,034	1,200,030	2.17	16.24
Recur	3,932,167	2,949,082	2,949,110	2.44	31.83
SumTo	15,003,637	8,002,122	17,002,921	2.29	76.61
Tak	572,495	333,954	572,488	2.31	8.16
Bubble	4,949,143	3,047,322	3,917,196	2.17	44.35
Detabify	2,323,037	1,295,022	1,673,021	1.92	26.92
IntMM	2,191,730	1,340,659	1,478,013	2.52	19.43
MergeSort	13,787,215	8,050,676	13,471,760	2.06	171.86
Perm	2,482,648	1,677,576	1,617,831	2.33	30.05
Puzzle	21,704,318	12,577,749	17,269,270	5.12	256.93
Queens	1,873,879	1,060,564	1,543,504	2.86	17.25
QuickSort	2,394,931	1,481,793	1,791,956	2.17	21.85
QuickSort2	3,974,032	2,702,648	2,756,506	2.56	33.90
Sieve	6,086,477	3,723,942	4,975,860	2.15	45.68
Towers	2,029,491	1,342,206	1,145,272	2.41	18.627
Tree	1,667,752	620,558	1,891,346	1.97	34.21
CParser	4,146,255	2,544,773	2,529,170	4.76	31.79
DeltaBlue	2,200,062	1,250,828	1,443,229	2.59	39.95
Diff	9,639,507	5,766,326	6,385,795	4.72	144.10
SParser	319,256	185,806	173,674	5.15	9.01
PrimMaker	2,331,941	1,291,980	1,516,633	4.67	50.80
Richards	8,504,293	4,741,143	5,908,623	2.06	116.31
RSA	212,595,415	124,215,649	157,813,545	5.56	n/a

Table A-1. Detailed data for unoptimized programs

Benchmark	calls	static calls	dispatches	avg. arity	time
AtAllPut	2	1	400,003	1.33	1.98
Recur	46,802	46,801	32	1.00	0.54
SumTo	2	1	4,000,402	1.24	3.73
Tak	63,622	63,616	238,540	1.42	0.42
Bubble	30	20	1,901,670	1.49	2.02
Detabify	1,000	1,000	446,003	1.27	0.71
IntMM	1,660	1,614	1,076,377	1.73	2.94
MergeSort	71,978	31,952	7,089,219	1.53	7.03
Perm	43,752	43,587	821,755	1.51	0.99
Puzzle	45,434	21,816	9,812,253	1.65	8.29
Queens	5,822	5,765	659,212	1.53	0.76
QuickSort	4,455	4,446	1,012,541	1.57	1.13
QuickSort2	4,441	661	1,504,432	1.55	1.52
Sieve	313,802	11	2,233,812	1.47	2.41
Towers	65,576	65,570	1,112,289	1.97	1.36
Tree	80,681	75,674	916,537	1.53	3.14
CParser	238,823	71,281	1,364,369	1.86	6.06
DeltaBlue	43,137	16,400	639,298	1.49	1.88
Diff	137,120	122,525	2,276,197	1.33	8.54
SParser	12,083	7,696	70,727	1.40	1.34
PrimMaker	113,874	16,697	532,530	1.40	7.73
Richards	217,612	29,454	1,824,197	1.43	3.66
RSA	6,968,500	5,508,500	40,675,365	1.27	167.52

Table A-2. Detailed data for type feedback

Benchmark	calls	static calls	dispatches	avg. arity	time
AtAllPut	1	-	3	1.00	2.06
Recur	46,810	46,800	-	1.00	0.54
SumTo	100	-	-	1.00	3.98
Tak	63,610	63,609	-	1.00	0.32
Bubble	2	2	885,120	1.23	1.85
Detabify	1,000	1,000	65,003	1.04	0.52
IntMM	1,606	1,606	326,402	1.22	1.88
MergeSort	71,915	71,914	1,636,111	1.12	5.52
Perm	43,307	43,307	201,622	1.12	0.87
Puzzle	21,379	21,353	4,833,156	1.39	7.24
Queens	5,751	5,700	323,204	1.32	0.70
QuickSort	4,436	4,436	446,510	1.25	1.00
QuickSort2	627	626	533,312	1.19	1.19
Sieve	10	10	81,923	1.03	1.25
Towers	65,561	65,561	751,748	1.68	1.60
Tree	80,665	75,665	594,582	1.32	2.99

Table A-3. Detailed data for TI-int

Benchmark	calls	static calls	dispatches	avg. arity	time
CParser	394,647	149,303	859,453	1.88	4.80
DeltaBlue	114,622	20,044	207,633	1.18	1.57
Diff	203,030	125,925	298,296	1.05	6.45
SParser	7,366	4,941	23,747	1.17	1.23
PrimMaker	70,202	21,229	102,888	1.08	6.36
Richards	217,502	29,430	1,338,814	1.28	3.56
RSA	N/A	N/A	N/A	N/A	N/A

Table A-3. Detailed data for TI-int

Benchmark	calls	static calls	dispatches	avg. arity	time
AtAllPut	1	-	400,004	1.42	1.63
Recur	46,810	46,800	33	1.45	0.54
SumTo	100	-	6,000,603	1.29	3.98
Tak	63,610	63,609	302,148	1.64	0.80
Bubble	510	2	2,659,018	1.71	3.32
Detabify	1,000	1,000	384,003	1.23	0.73
IntMM	1,606	1,606	1,196,414	1.85	2.43
MergeSort	149,843	84,202	7,095,791	1.57	12.80
Perm	68,507	68,507	908,153	1.59	1.88
Puzzle	2,230,997	21,353	12,314,671	1.79	52.18
Queens	5,751	5,700	772,304	1.64	1.35
QuickSort	4,436	4,436	1,148,074	1.75	2.86
QuickSort2	24,862	627	1,662,329	1.71	4.73
Sieve	10	10	1,895,853	1.40	2.63
Towers	65,561	65,561	1,193,963	2.16	1.66
Tree	80,665	75,665	815,871	1.52	3.50
CParser	402,583	149,303	1,229,806	1.82	5.24
DeltaBlue	131,581	20,045	498,893	1.40	2.69
Diff	207,925	125,924	1,854,196	1.30	8.91
SParser	7,546	5,001	54,829	1.36	1.42
PrimMaker	73,888	23,332	289,021	1.20	7.01
Richards	217,502	29,430	1,601,904	1.32	3.87
RSA	7,066,669	5,510,379	31,928,807	1.22	203.09

Table A-4. Detailed data for TI

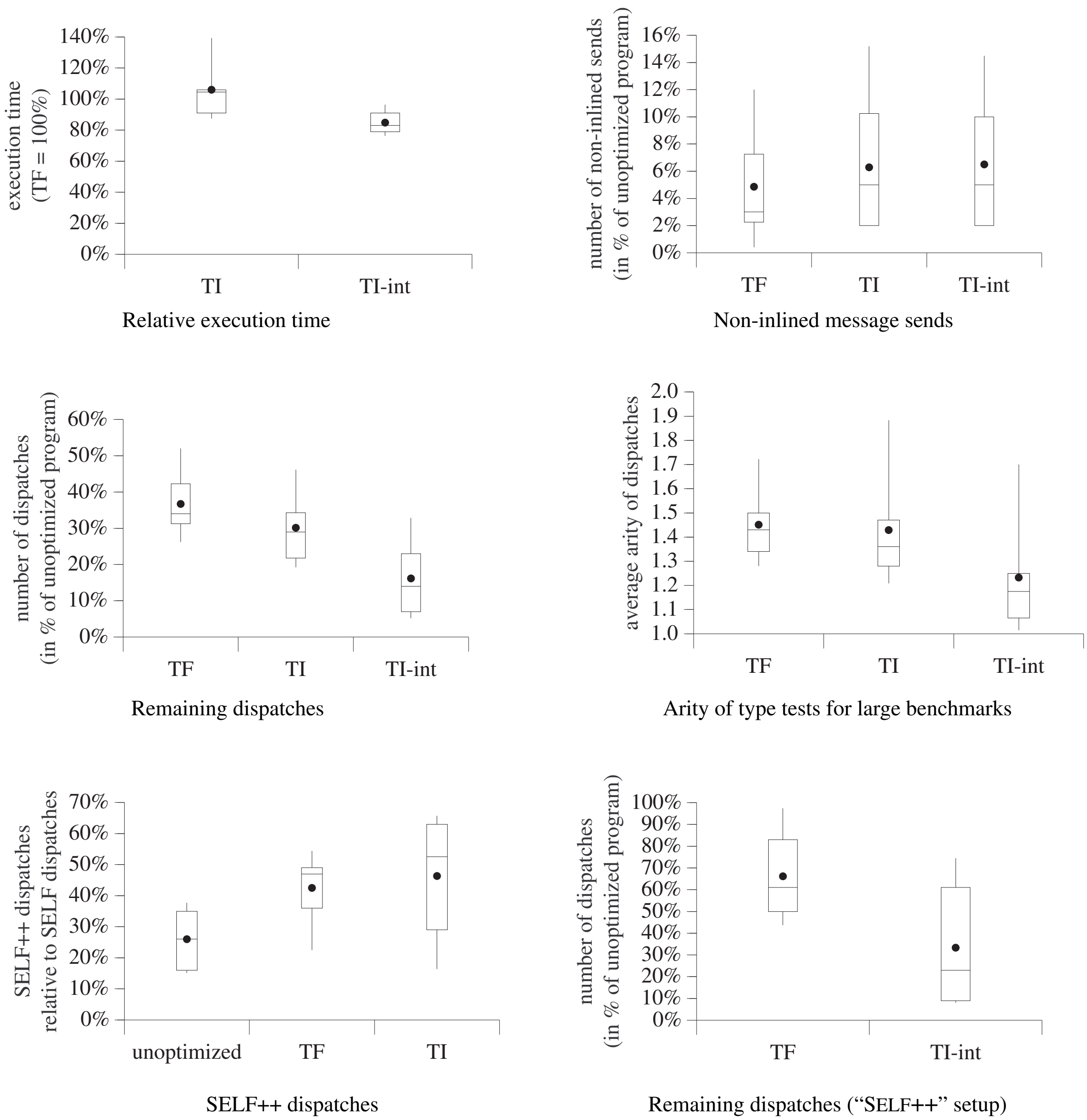

Figure A-1. Box-chart summaries of data for large benchmarks

Box charts show the range of data (vertical lines) as well as the 25% and 75% percentiles (end of the boxes) and the median (horizontal lines). The mean is indicated with a dot (•).