Training Compilers for Better Inlining Decisions

Jeffrey Dean and Craig Chambers

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

Technical Report 93-05-05
May 1993

Training Compilers to Make Better Inlining Decisions

Jeffrey Dean and Craig Chambers

Technical Report 93-05-05
May 1993

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195
{jdean,chambers}@cs.washington.edu

Abstract

Optimizing implementations for object-oriented languages rely on aggressive inlining to achieve good performance. Sometimes the compiler is over-eager in its quest for good performance, however, and inlines too many methods that merely increase compile time and consume extra compiled code space with little benefit in run-time performance. We have designed and implemented a new approach to inlining decision making in which the compiler performs inlining experimentally and records the results in a database that can be consulted to guide future inlining decisions of the same routine at call sites that have similar static information. Central to our approach is a new technique, called *type group analysis*, that calculates how much of the static information available at a call site was profitably used during inlining. The results of type group analysis enable the compiler to compute a generalization of the actual static information for a particular experiment, significantly increasing reuse of database entries. Preliminary results indicate that compile time is almost cut in half with only a 15% reduction in run-time performance.

1 Introduction

Much of the performance gap between pure object-oriented languages and more traditional languages may be attributable to the higher frequency and greater expense of procedure calls in object-oriented languages. In pure object-oriented languages, all operations are conceptually implemented as message sends, including such basic operations as arithmetic, control structures, and variable accesses. Implemented straightforwardly, a message send requires additional run-time type tests or memory indirections on top of a normal procedure call. Consequently, efficient implementations of object-oriented languages focus on implementing message sends more efficiently.

SELF is a dynamically-typed object-oriented language that provides only object-oriented features to the programmer [Ungar & Smith 87, Hölzle et al. 91a]. In order to obtain reasonable performance, the SELF compiler performs several optimizations that allow a significant percentage of message sends to be inlined [Chambers & Ungar 91, Hölzle et al. 91b, Chambers 92]. Once the compiler has the option of inlining a method, it must make a decision about whether it is beneficial to inline the method, based on the estimated trade-offs of increased compile time and compiled code size versus improved run-time performance. The improvement in run-time performance is attributable both to the direct benefits of eliminating the procedure call and to the indirect benefits obtained by optimizing the inlined routine in the context of the calling routine. In many cases, the indirect benefit is greater than the direct benefit, particularly for control structures. Balancing the

costs against the benefits to simultaneously achieve short compile times and short execution times is a difficult task. Keeping compilation time low is particularly important for SELF since SELF is based on *dynamic compilation* technology [Deutsch & Schiffman 84, Chambers *et al.* 89], where the system interleaves compilation and execution, thereby making compilation pauses visible at run time. On the other hand, improving run-time performance through inlining is crucial to SELF since all computation and control structures are implemented through message passing and performance would be abysmal without inlining.

The inlining decision-making heuristics in the current SELF compiler are based on a quick, superficial estimate of the source length of the candidate method. These heuristics were tuned in such a way that they tended to err on the side of inlining too much, in order to provide high levels of run-time performance. As a consequence, compilations of large programs could be quite lengthy. Furthermore, the parameters used in the current heuristics were based on the specific source code used in the current version of the SELF system, and they do not adapt easily to seemingly superficial changes to the source as the SELF system evolves.

We have developed an alternative strategy to making inlining decisions that is more effective and more adaptive to change than the current heuristics. Our compiler bases its inlining decisions on a database of space cost and run-time benefit information associated with candidate methods. Whenever the compiler needs to make an inlining decision, it consults this database to get relatively accurate cost and benefit information. If no entry in the database covers the call in question, the compiler inlines the target method experimentally to compute the compiled code space taken and expected run-time saved as a result of the inlining. While the compiler is optimizing the experimentally-inlined code, it applies *type group analysis* to calculate how much of the static information available at the call site was profitably used to perform optimizations of the callee. When the experiment is completed, an entry is added to the database characterizing the time and space benefits and costs of inlining the method for call sites with a certain amount of available static information, to be used at future compilations of calls to the same method. In effect, the compiler trains itself to make more informed inlining decisions. Early measurements of our new system indicate that compile time is cut nearly in half, with only about 15% degradation in run-time performance.

Section 2 describes the existing heuristics for making inlining decisions and introduces two examples to point out some limitations with the approach. Section 3 describes our new strategy of using a database of past inlining decisions to guide similar inlining decisions in the future. Section 4 reports on early performance results comparing the heuristic-based approach with the inlining database approach. Section 5 outlines some future work and describes some other areas where the new techniques could be applied. Section 6 summarizes related work.

2 Inlining Using Source-Level Length Estimates

Inlining compilers must avoid inlining routines which are “too long,” where the expected run-time improvement due to inlining is relatively small and the expected cost in compile time and compiled code space is relatively large. (Inlining compilers must avoid other hazards, too, such as inlining a recursive routine indefinitely, but these other hazards are ignored in this paper.) Existing compilers, including the current SELF compiler, determine whether a method is “too long” to inline profitably by scanning its source code and computing some estimate of its length. This estimate is effective to the extent that it correlates well with the actual compiled code size and compile time taken to

inline the routine. If the indirect benefits of inlining are relatively minor, then the estimate is likely to be good. If, however, the inlined routine can be substantially optimized after inlining, the superficial source-level estimate is likely to perform poorly.

The current SELF length estimate is merely a count of the number of message sends appearing in the target method that do not correspond to local variable or instance variable accesses. Messages appearing in the bodies of closures (blocks in SELF parlance) nested within the method are also included, under the assumption that most closures will end up being inlined. Once the length of a target method is estimated, it is compared against a threshold value, and the method inlined if its length does not exceed the threshold.*

We will illustrate these points using two examples. The following code calls the `to:By:Do:` method to get the effect of a `for` loop:

```
...
1 to: 100 By: 2 Do: [ body of the loop ].
...

"integer" to: end By: step Do: block = (
   step compare: 0
      IfLess: [ to: end ByNegative: step Do: block ]
      Equal: [ ^error: 'step is zero in to:By:Do: loop' ]
      Greater: [ to: end ByPositive: step Do: block ] ).

"integer" to: end ByPositive: step Do: block = ( | i |
   i: self.
   [ i <= end ] whileTrue: [
      block value: i.
      i: i + step.
   ].
   nil ).

"closure" whileTrue: block = (
   [ value ifFalse: [^ nil].
     block value.
   ] loop ).
```

When compiling the initial call to `to:By:Do:`, the current SELF system considers the `to:By:Do:`, `compare:IfLess:Equal:Greater:`, `to:ByPositive:Do:`, `whileTrue:`, `loop`, `ifFalse:`, `+`, `<=`, and several other user-defined methods in turn and finds each of them suitable for inlining. When the smoke clears, the compiler has generated code for the original `to:By:Do:` call that contains no message sends and closely resembles the code generated by a traditional compiler for a built-in `for` language construct. This sizable compression of source code to compiled code occurs as a result of the indirect benefits of being able to perform optimizations after inlining, particularly because the constant `step` argument allows the `compare:IfLess:Equal:Greater:` message to be constant-folded, eliminating two of the three cases for a for loop. If the `step` argument had not been a compile-time constant, then the compiled code costs would have been much greater, and it would have been less clear that inlining `to:By:Do:` was profitable.

* In fact, there are two thresholds used in the current SELF compiler: one for a normal method and another, higher, threshold for a method called with a closure argument. A higher threshold is used to preferentially inline methods that appear to be control structures.

The following `advance` method is taken from a parsing program. It moves the current character being scanned forward to the next character and updates some local state of the parser.

```
"parser" advance = (
   eof ifTrue: [^error: 'advancing past end'].
   inputPosition: inputPosition successor.
   setThisChar.
   self ).

"parser" eof = ( inputPosition > input lastKey ).

"parser" setThisChar = (
   whatWasExpected: ''.
   printThisChar.
   thisChar: (eof ifTrue: [' ' first] False: [input at: inputPosition]).
   self ).
```

According to the same length estimates that identified the `to:By:Do:` method as a good method to inline, the `advance` method looks profitable to inline. So the compiler inlines it, as well as the `eof`, `setThisChar`, and other methods. Unfortunately, inlining `advance` improves run-time performance only slightly, because the non-inlined version of `advance` would run nearly as fast as one inlined into its caller; no substantial indirect benefits of inlining accrue in this case. But the compile-time and compiled code space costs are substantial. A better choice would have been not to inline the `advance` method. This is a real problem in our parser application, where 25 separate calls to `advance` each get their own inlined copy of this routine (and the inlined routines that it calls).

The length estimate and other heuristics could be refined further, to attempt to distinguish between cases like `to:By:Do:` and cases like `advance`, and we pursued this path briefly. However, it became increasingly difficult to improve the length estimates in such a way that the compiler made substantially better decisions about what to inline. The more parameters and superficial metrics considered by the estimator, the more complicated the weighting functions and cut-off thresholds become, and the more sensitive the estimator becomes to small details of the source code.

3 Inlining Using a Database

To make better inlining decisions, the compiler needs better information on the actual costs and benefits of inlining a routine in the context of a particular call site. If the compiler knew the compiled code space costs and expected run-time improvements of inlining a particular routine at a particular call site, it could make much more informed decisions about what is profitable to inline, and the compiler would be less sensitive to superficial source-level properties of the candidate routines.

One method for obtaining more accurate cost and benefit information would be to actually go ahead and experimentally inline the candidate routine, optimize it in the context of the call, and examine the resulting code to determine its compiled code space cost and estimate its savings in execution time as a result of inlining. If the trade-offs of increased code space for improved performance were reasonable, the inlined code would be preserved; otherwise, the compiler could "abort" the inline expansion and revert back to a non-inlined procedure call. If coupled with good decision-making heuristics balancing compiled code space costs and execution speed improvements, this "brute force" approach could make much better decisions than the current source-level length estimate-based heuristics. Of course, compilation time costs would not be

reduced in this approach because the compile time to inline the routine is expended even if the inlining is later aborted.

The key idea to our new strategy for making inlining decisions is to couple the brute force approach with a long-lived database in which to store the results of inlining experiments. Future attempts to inline the same method at similar call sites would consult the database instead of repeating the inlining experiment, thereby amortizing the extra cost of the experiment over all uses of the information in the database. If a method could be inlined from many similar call sites, the amortized cost of the one brute force experiment would be fairly small, approaching that for the simple source-level heuristics. Furthermore, if a few methods can be identified that turn out to be bad choices to inline, the savings reaped by not inlining those methods pays the cost of the experiments many times over. Our experience in SELF is that many routines, such as control structures and operations on standard data structures, are invoked from multiple call sites. Additionally, SELF's dynamic compilation strategy often purges least-recently-used compiled methods from its internal cache, and regenerating the compiled code when it is next needed causes the compiler to revisit the same inlining decisions again.

Section 3.1 describes the information stored in the database in more detail. Section 3.2 explains how the information in the database helps the compiler make better inlining decisions. Section 3.3 describes how the database is filled through inlining experiments. Section 3.4 addresses complications arising from closures.

3.1 Inlining Database

The database stores the results of past inlining experiments and is used to guide future inlining decisions. Each database entry contains two parts: a key, identifying call sites to which this entry applies, and experiment result data, recording the costs and benefits of inlining the routine at such a call site. The key can be broken down into two components, both of which must match for the database entry to apply:

- A "message lookup key" used by the compiler to index into the method lookup cache. This information uniquely identifies the target method being inlined, and contains information such as the name of the message and the type of the receiver.
- A description of the amount of static type information known at the call site about the receiver and arguments to the message, represented by *type groups*. Type groups are described in more detail below.

The experiment result data stored in each database entry includes:

- an estimate of the compiled code space cost of inlining the routine;
- an estimate of the time the inlined code will take to execute; and
- an estimate of the execution time saved due to inlining the routine, including both the direct and the indirect savings.

Both time taken and time saved fields are included so that inlining decisions may depend on both the relative and the absolute execution speed benefits of inlining.

The indirect benefits of inlining a method are determined largely by the amount of static type information available at the call site about the arguments to the inlined routine. If some information is available at the call site, such as that the `step` argument to `to:By:Do:` is an integer constant, then the inlined version of the routine may be able to be optimized. A call site with less static

information, such as one where the step was a computed value of unknown type, might not be able to perform as many optimizations after inlining, and so the costs of inlining would be greater and the benefits less.

Type groups are our mechanism for summarizing the extent to which the static information available about each argument impacted the costs and benefits of inlining the routine. If the code generated as a result of inlining depends on some static information about an argument, the type group for that argument records that this static information is needed. Similarly, if the generated code was not able to be optimized well because some static information about an argument was not available, the type group for that argument will record the lack of static information as well. The same target method may appear in multiple database entries, with each entry corresponding to different amounts of static type information available to the call site and having different costs and benefits to inlining. Being able to distinguish call sites in this way is crucial to being able to accurately assess the indirect effects of inlining in the context of the call site.

The kinds of type groups that need to be represented depend on the kinds of optimizations the compiler can perform. For the SELF compiler, we support the following kinds of type groups:

Type group	**Description**
UniversalTypeGroup	any type
AMapTypeGroup	any type whose class is known (“map” in SELF parlance)
AConstantTypeGroup	any compile-time constant
ABlockTypeGroup	any closure (“block” in SELF parlance)
SubsetTypeGroup(*T*)	any type that is contained in the type *T*
IntersectionTypeGroup(G_1,G_2)	any type that is in both type group G_1 and type group G_2
DifferenceTypeGroup(G_1,G_2)	any type that is in type group G_1 but are not in type group G_2

For example,

IntersectionTypeGroup(AConstantTypeGroup, SubsetTypeGroup(Integer))

describes an argument that is known to be some integer constant, and

DifferenceTypeGroup(UniversalTypeGroup, AMapTypeGroup)

describes an argument whose class is not known statically.

The following are some of the entries computed by our system for the `to:By:Do:` method and the `advance` method, illustrating the sort of information typical of database entries (type group information is summarized in parentheses between each argument):

	Data		
Key	**Run time taken**	**Run time saved**	**Space taken**
(integer) to: (integer) By: (integer constant) Do: (a block)	99	1149	25
(integer) to: (integer) By: (integer, not constant) Do: (a block)	100	923	91
(integer) to: (not known to be an integer) By: (integer constant) Do: (a block)	136	668	73
(integer) to: (not known to be an integer) By: (integer, not constant) Do: (a block)	343	873	109
(parser) advance	45	16	86

3.2 Using Database Information to Make Inlining Decisions

Using the database to make inlining decisions is relatively straightforward. Given a call site to a particular routine, the database is queried to find entries for that routine, i.e., entries with matching message lookup keys. The static type information known at the call site about the receiver and arguments is then tested for inclusion in the corresponding type groups of each database entry for the target routine.* If a matching entry is found, the compiler extracts the cost/benefit information out of the database key and makes a final inlining decision. If a matching entry is not found, the compiler may elect to perform an inlining experiment, as described next in section 3.3.

To make a final inlining decision, the compiler compares the estimated execution time for the inlined version of the routine to the estimated execution time of the routine if it were called out-of-line (computed by summing the time taken and the time saved fields in the database entry). The compiler also compares the compiled code space cost of the inlined version of the routine to the space cost of a single message send.† Finally, the decision function takes into account additional information about the call site, such as its expected execution frequency within the calling routine. Given all this data, a decision function makes a yes or no decision on whether to inline the function. Designing a good decision function is still a hard problem, but at least with the database information it has more accurate information upon which to base the decision. We have implemented a simple, ad hoc function that appears to work adequately in the cases we have tested, but more work in this area is possible.

3.3 Inlining Experiments

If no matching database entry is found when the compiler is trying to decide whether to inline a routine, the compiler has the option of doing an experimental inline-expansion. An inlining experiment is treated just like a normal inline-expansion, except that extra bookkeeping nodes are inserted in the control flow graph to mark the beginning and end of the inlined routine, and the compiler monitors its activities to calculate the following summary information:

- estimates of the expected execution time and compiled code space costs of the inlined code after optimization,
- an estimate of the execution time saved as a result of inlining the routine, and
- the type groups for the receiver and arguments.

At present, we do not directly measure the compilation time costs of the inlining experiment. Instead, we compute the compiled code space costs, which in our experience tend to be proportional to compilation time and are more reliable and reproducible.

The next three subsections, 3.3.1 through 3.3.3, explain in more detail how these three kinds of summaries are computed. Subsection 3.3.4 explains what happens when an experiment is concluded. Nested experiments are addressed in subsection 3.3.5.

---

* The type group information of different database entries for the same routine is disjoint, so at most one database entry will match any given call site.

† We assume that most methods will also have an out-of-line version compiled for other non-inlined call sites, so we do not take this space cost into account as part of the inlining decision.

3.3.1 Calculating Space Cost and Expected Execution Time

In order to calculate the estimated run-time and the compiled code space cost of an inlining experiment, each type of node in the control flow graph is assigned a time cost and a space cost. Time costs are estimates of the number of machine cycles taken to execute the node, and space costs are estimates of the number of words of machine code generated for the node. The table below indicates the costs assigned to each type of node:

Operation	**Space cost**	**Time cost**	**Notes**
Arithmetic & branch nodes	1	1	Single machine instruction
Load & store nodes	1	2	
Primitive calls	5	15	Time cost is an estimate of "the minimum amount of time" a non-inlined primitive call or message send takes to execute
Message send	7	20	
Block clone nodes	5	9	Cost of creating a closure
Bookkeeping nodes	0	0	Generate no instructions

Calculating the space cost of an experiment is straightforward: we simply sum the estimated space costs for each node within the experiment. The space cost is not entirely accurate because the inlining phase of the compiler runs prior to register allocation and instruction scheduling, and additional instruction generating nodes (such as register moves) may be inserted into the flow graph during these phases; to some extent the space costs attributed to each node include the expected amount of register move and other support instructions likely to be added in later.

Computing the estimated run-time for a routine is a bit more difficult. We cannot simply sum the time costs for the nodes, because nodes may not execute exactly once (due to loops and branches) each time the inlined routine is called. Instead, we weight the time cost of each node by the number of times we expect it to be executed for a particular invocation of the inlined function. Our execution frequency estimation rules are standard. Successors of branch nodes are each given a weight equal to half the weight of the branch node (unless the compiler has information that one of the branches is much more likely than the other, such as a branch which tests for arithmetic overflow). Merge nodes sum the weight of their predecessors. Nodes within a loop are given a weight equal to ten times the weight entering the loop. We calculate the expected execution time of the inlined routine as the weighted sum of the time cost of the nodes in the experiment, after optimization.

Assigning a time cost to non-inlined message send nodes is difficult. The compiler has no good way of estimating the time spent inside an arbitrary function call. To resolve this dilemma, we pick a value that corresponds to the typical length of a short function, such as one that just accesses an instance variable. We choose to err on the side of a smaller time cost estimate than might actually be the case so that this error does not cause some routine to look unprofitable to inline (the longer a routine is estimated to run, the smaller the relative time savings for inlining the routine). Primitive calls are treated similarly, although for some primitive calls the system could have better information about typical durations.

3.3.2 Calculating Time Saved

To compute the amount of execution time expected to be saved as a result of inlining, the compiler monitors the optimizations it performs based on static information available at the call site that would not be available if the routine were not inlined. Our current implementation concentrates

only on the most important optimizations, which are related to static type information about the arguments to the inlined routine. These optimizations include inlining of other messages based on class type information about the message receivers, eliminated run-time type tests and type checks also based on class type information, and constant folding based on compile-time constant information (treated as a very precise form of type information). Other static information at the call site might have an effect on the generated code, such as the set of available expressions for common subexpression elimination [Aho *et al.* 86], but we expect this to be a relatively minor effect.

Each time an optimization is performed, the compiler estimates the amount of time the optimization saved, derived from the code which would have been emitted and executed if the optimization had not been performed. It then adds this number, weighted by the execution frequency of the optimized nodes as computed for time taken, to a running sum of time saved.

As an example of the calculations performed to estimate time saved, consider the `to:By:Do:` method, when called with a `step` parameter which is known to be an integer. The `compare:IfLess:Equal:Greater:` message compares `step` to the integer constant 0. If the type of `step` were unknown, as it would be if the `to:By:Do:` method were not inlined, the compiler would have inserted a type test to determine if `step` was an integer. However, since the compiler knows statically that `step` is an integer, the type test is unnecessary and can be optimized away. An integer type test takes 2 cycles (one cycle for a tag comparison and one cycle for a conditional branch), so the unweighted amount of time saved by eliminating the type test is 2 cycles. This number is weighted by the expected execution frequency of the type test, which since it appears at the beginning of the code for `to:By:Do:` would be 1.

3.3.3 Determining Type Group Information

The final kind of information extracted during an inlining experiment is the type group information about the arguments to the inlined routine. The purpose of the type group is to describe how much of the type information available at the call site was actually used during optimizations of the inlined code, so that the database entry constructed for this experiment will apply to the largest possible number of call sites.

At the beginning of the experiment, all arguments are associated with the UniversalTypeGroup, indicating that, so far, no static information about the arguments has been used. As optimizations are performed that use some part of the static type information about an argument, the associated type group is narrowed by intersecting it with a type group that represents the kind of static information that enabled the optimization. For example, when optimizing the inlined `to:By:Do:` method, the first message considered is sending the `compare:IfLess:Equal:Greater:` message to the `step` argument. If for example the call site passed the constant `2` as the step argument, the `compare:...` message could be (and would be) inlined. Inlining was enabled by the fact that the argument was known to be an integer (inlining does not depend on the fact that the argument was a constant, though), and so the initial UniversalTypeGroup associated with the step argument would be intersected with SubsetTypeGroup(*integer*).

Complementing this “positive” type group information is “negative” type group information. If some static information about an argument was lacking, and a potential optimization could not be performed, then the argument’s type group information is again narrowed by intersection with a type group describing the absence of some type information that would have been useful. For

example, if in the `to:By:Do:` example the `step` argument was some unknown quantity of unknown type at the call site, the `compare:...` message could not be inlined. To record that the static type information was not precise enough to enable some important optimization, the DifferenceTypeGroup(UniversalTypeGroup, AMapTypeGroup) type group is intersected with the previous type group for `step` (in this case the UniversalTypeGroup) to record the lost opportunity.

Once an inlining experiment is complete, the type groups associated with each argument will record how much of the static information available at the call site either enabled or disabled optimizations during inlining. These type groups become part of the key of the database entry. Future opportunities to inline the same routine at other call sites will examine the type group information to determine whether they have similar amounts of static information available about arguments. If they do, then they are very likely to generate substantially the same code as did the inlining experiment, and so the summary experiment data associated with the database entry can be treated as being reasonably accurate. Positive type group information restricts database entries to other call sites which can also do the same optimizations. Negative type group information prevents call sites with lots of static information from incorrectly matching against a database entry where fewer optimizations were possible, thereby preventing a bad initial experiment from ruining the chances of future inlinable calls.

3.3.4 Finishing an Inlining Experiment

When all of the nodes in an experiment have been processed and optimized, the compiler computes the overall time saved, time taken, and space cost data and adds a new entry to the database. It also consults this new entry to decide whether the experiment should be committed or aborted, using the same tests for other inlining candidates described in section 3.2. If the decision is to abort the experiment and undo the inline-expansion, the compiler splices out all the control flow graph nodes in the experiment and reinserts a non-inlined message send.

3.3.5 Nested Inlining Experiments

If, during the course of an inlining experiment, the compiler encounters an inlinable message without a corresponding entry in the database, it enters a nested experiment. The nested experiment is handled just like a normal experiment, being careful to isolate any effects of running the nested experiment from affecting the enclosing experiment until a decision is reached about whether to commit or abort the nested experiment. If the nested experiment is committed, then any time and space costs associated with the nested experiment are propagated to the enclosing experiment. If the nested experiment is aborted, then all time and space date associated with the nested experiment are thrown away, and instead the time and space cost of a non-inlined message send is added to the enclosing experiment. This protocol is designed so that the effect of a nested experiment is the same as if a matching database entry had been available.

3.4 Closures

Closures are used frequently in SELF code, as arguments to user-defined control structures, as exception handlers, and as mechanisms to communicate data from callee to caller. Each distinct static occurrence of a closure expression is assigned its own unique class, to represent its specific source code and enclosing lexical context format. If closures were treated just as any other object, many separate database entries would be generated for routines with closure arguments, each with a different SubsetTypeGroup(*a block class*) type group associated with the closure argument. For

example, each occurrence of a `to:By:Do:` message typically passes a different argument block (the body of the loop), and each of these calls would end up with different type group information. As a result, there would be little reuse of database entries for routines taking closures as arguments, thus defeating much of the purpose of constructing the database in the first place.

To solve this problem, we factor out the details of particular closure arguments from database entries. If, during an inlining experiment, the body of a closure is inlined (by inlining a send of `value` to the closure), instead of intersecting SubsetTypeGroup(*closure type*), the compiler intersects the generic inlined closure type group ABlockTypeGroup; this type group will match any closure type. When constructing the database entry for a routine with an argument type group including ABlockTypeGroup, the compiler subtracts off the time and space cost and time saved data attributable to the body of the closure from the data stored in the entry. This is done by running an experiment on all `value` messages sent to closures within an experiment, and subtracting off the results of these experiments from all enclosing experiments to which the closure was passed as a parameter. A database entry thus contains the time and space costs and time saved data for a control structure itself "surrounding" the closure body, and does not include the effects of closures which were passed as arguments to the routine.

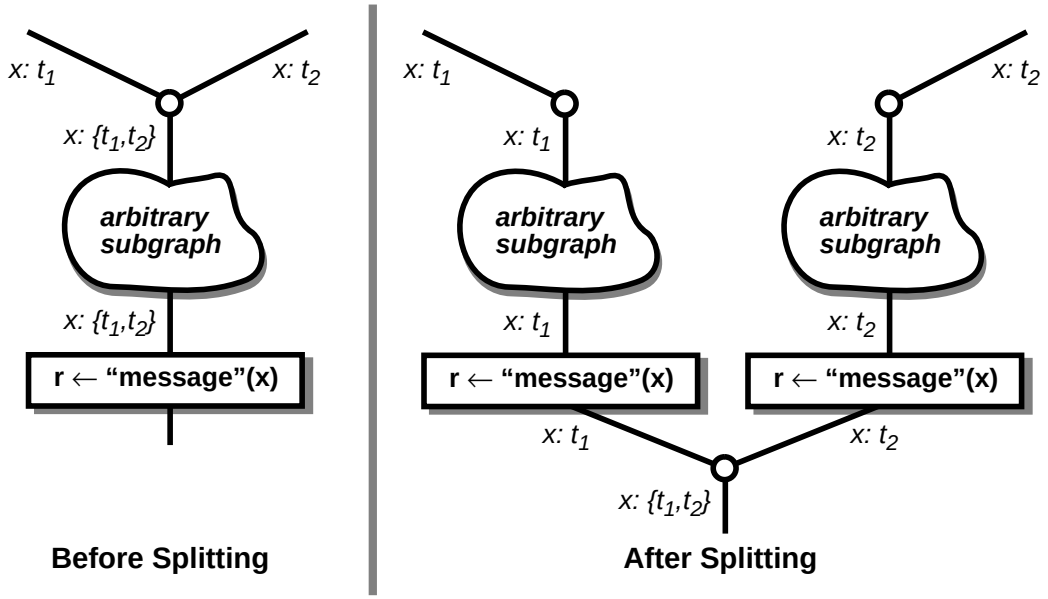
Ideally, when a client of the database matches an entry with an argument type group including ABlockTypeGroup, it would add in the corresponding time and space data for the body of the closure to that stored in the database entry to compute the complete cost/benefit information for the call. Unfortunately, accurate time and space data for the closure is difficult to obtain without performing an experiment whose results are not likely to be reused. The inlined space cost for a closure can be argued to be zero, because, unlike normal routines, no out-of-line compiled version of the closure body is expected to exist if the closure is inlined instead. Time taken and time saved metrics are less easily dispensed with. Lacking a better solution, our current implementation simply treats these values as zero, also, reducing the inlining decision to one based solely on the called routine ignoring the impact of the closure.

By ignoring the effect of closure arguments when determining whether to inline some routine, the compiler uses just the cost/benefit trade-offs of the routine excluding the closure argument when making decisions about inlining profitability. For common cases like control structures, where the structure itself is relatively small and quick, inlining is deemed profitable, which is usually the right decision. Routines that take closure arguments as exception handlers typically are relatively large and time-consuming, so our heuristics judge such a method a poor inlining candidate; again, this usually is the right decision. Even though our treatment of closures is not ideal, in practice it appears that our techniques make reasonable decisions. Improving the techniques for handling closures would be a good area for future refinement, nevertheless.

3.5 Splitting

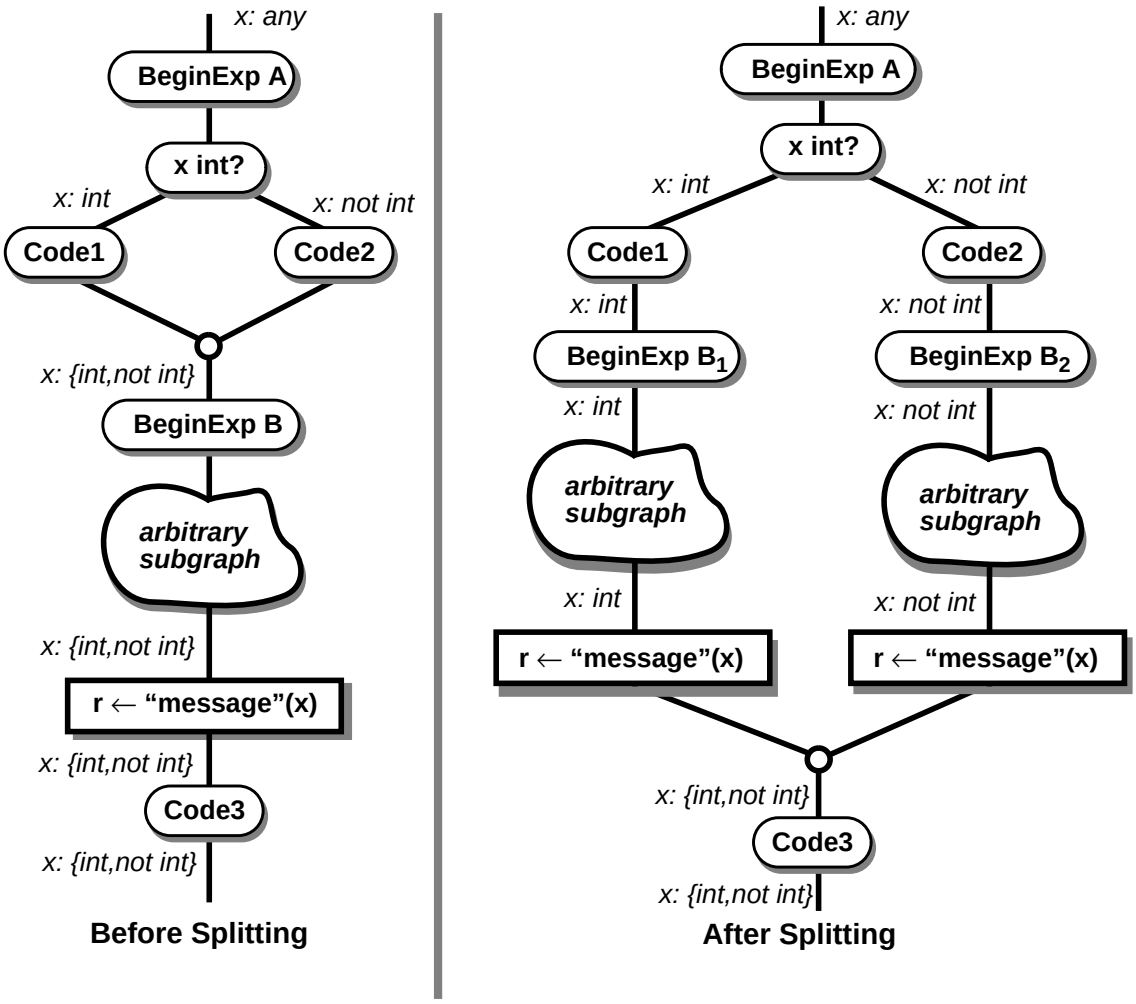
Splitting refers to the duplication of CFG nodes across multiple paths, rather than merging the paths together to form a single path. It is done to preserve type information present along the paths that would be lost if the paths were merged together. This allows transforming *polymorphic* code along the single path into multiple paths containing *monomorphic* code, enabling optimizations such as inlining to be performed along the paths [Chambers 92]. Splitting occurs when the compiler has merged together multiple paths in the CFG graph, but later decides that it would be worthwhile to postpone the merge so that it can exploit type information on one or more of the merged paths. The presence of splitting causes major headaches throughout the compiler, because code which has

been processed under one set of assumptions must be split apart to act as if it was compiled separately under different assumptions, and the experiment process does not escape these difficulties. The general splitting situation is illustrated below:



The difficulty is that the arbitrary subgraph being split can contain **BeginExp** nodes. Splitting one of these nodes requires splitting a single experiment into multiple experiments, each with (possibly) narrowed type information present for the actual parameters. We can no longer assume that each experiment has a single enclosing experiment. This complicates the propagation of time

and space information from inner experiments to enclosing experiments. The following CFG fragment illustrates the problem:



Without reluctant splitting, we could accumulate the time and space costs and time saved information for an inner experiment, and propagate this information to enclosing experiments when the inner experiment has been completed. With splitting this approach is complicated because partially completed experiments can be split and we need to avoid double counting time and space data when we propagate information from split experiments to enclosing experiments. This is the situation illustrated above. The inner experiment B has been split into two experiments, B_1 and B_2, but the enclosing experiment A was not split. This presents problems because we want to avoid double counting nodes in the inner experiment when we combine the results of the two inner experiments with the outer experiment A.

To make this problem more concrete, consider the nodes represented by **Code3** above. Assuming no further splitting takes place, we want to count the time and space costs of these nodes in both of the (now split) inner experiments B_1 and B_2, but want to count their costs only once in the outer experiment A. Our solution is to add in time and space costs for each CFG node simultaneously to all of the experiments which enclose the node, rather than waiting until the inner experiments have ended and trying to propagate the information from these inner experiments to their enclosing

experiments. Each experiment maintains its own view of the innermost experiment's time and space costs and type group information. When an inner experiment is finished, a decision is made to either keep or abort the experiment in the same way that the database information is used to make inlining decisions (see section 3.2). If the decision is to keep the inner experiment, the data accumulated in each enclosing experiment is combined with the existing experiment data in each experiment. If the decision is to abort the experiment, then the accumulated information stored within each enclosing experiment is discarded.

4 Performance

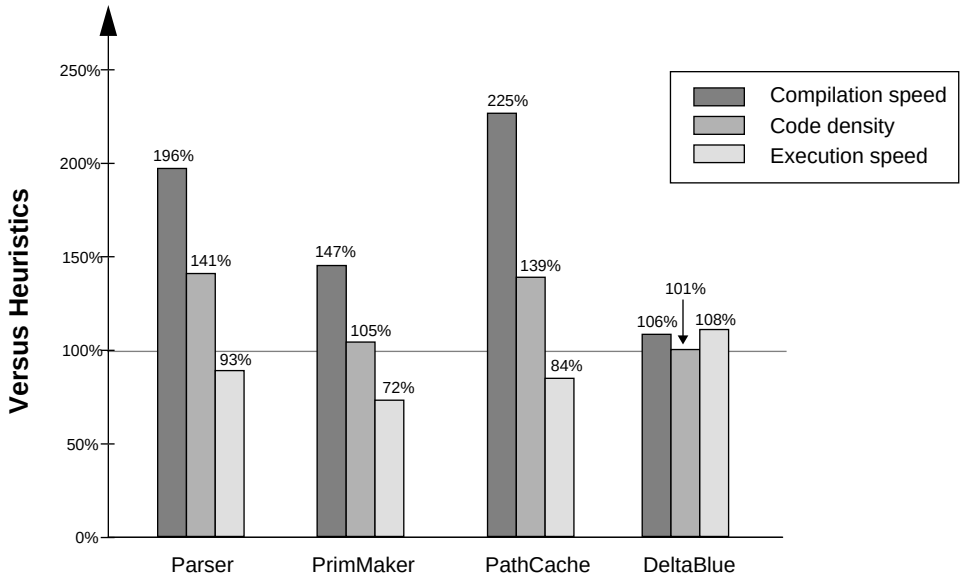
To evaluate the effectiveness of our approach, we measured the compilation speed, execution speed, and code density of a number of benchmark programs using both the existing source-level inlining heuristics and our new experiment-driven approach. The source-level heuristic used as a baseline was the best source-level length estimate-based heuristic we were able to concoct, as described in section 2, which is now running in the standard SELF system.

We first compared the two inlining decision-making approaches for a suite of small benchmark programs (including the Stanford integer benchmarks and some very simple benchmarks consisting of only a few loops). These benchmarks showed very little difference between the two approaches for any of the metrics. We had expected this, because the original inlining heuristics had been developed with these smaller benchmarks as test data, and so the existing heuristics had been tuned to always make the right decisions for this code. We were reassured that the new experiment-driven approach kept up with the existing heuristics, also making the right decision.

We then compared the two approaches for some larger, more object-oriented programs, where we suspected that the existing source-level heuristics were making some poor decisions. It was for these programs that we hoped compile times could be improved without significantly affecting run-time performance. We examined four programs:

- `parser`, a 400-line parser for an old version of SELF syntax;
- `primMaker`, a 1300-line program that generates external primitive wrapper functions from an interface description file;
- `pathCache`, a 270-line program that traverses the object graph and assigns path names to objects; and
- `deltaBlue`, a 600-line incremental constraint solver.

The following chart reports the compilation speed, execution speed, and code space density of these four benchmarks for our new system, relative to the standard system.* (Appendix A includes the raw data.) Bigger bars indicate better performance for our new system.



On average, compilation speed increased by a factor of 1.5, code space usage shrunk by 20%, and run-time performance dropped by 15%. `deltaBlue` even showed a slight increase in run-time performance. For most of these benchmarks, the new experiment-driven inlining approach clearly is a significant improvement over the existing source-level length estimate approach. The reduction in execution speed for `primMaker` is the weakest result. We suspect that this is caused by incorrectly deciding not to inline a method used in linked-list iteration, and we are working on improving the accuracy of the data collected during experiments to resolve this problem.

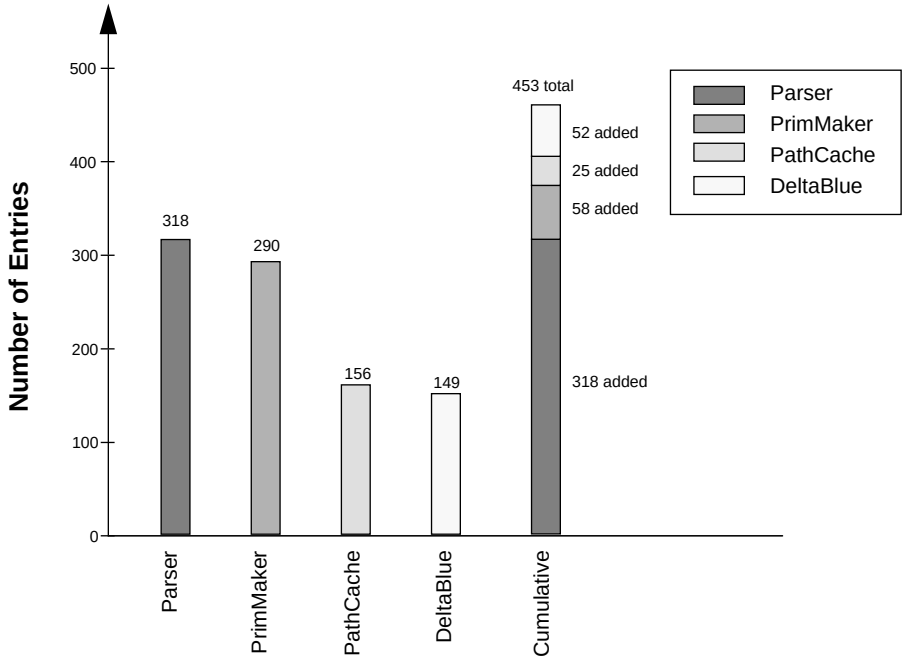
The compilation speed, execution speed, and code space density numbers above are with a full database (i.e., for compiling the program a second time reusing the database filled from an earlier execution of the benchmark). When compiling a program for the first time with an empty database, compile time still decreases on average, although not as dramatically as with a filled database. Appendix A includes the raw data for both the primed and the empty database cases.

In compiling the four benchmark programs, the inlining database was queried to make an inlining decision 7486 times. In 7240 of these cases, the decision was made in favor of inlining the method. In only 3% of the cases was it decided that the method should not inlined. That so few methods could have such a dramatic affect on compilation performance was somewhat surprising to us. This underscores the importance of making accurate inlining decisions; even a few incorrect decisions can significantly degrade compilation speed and, to a lesser degree, code space usage.

The size of the inlining database itself is important for judging the practicality of our approach. The following graph reports the number of database entries generated by compiling each of the four benchmark programs. The first four bars are the number of entries generated by starting with an

* The values in the chart are calculated as the compilation time, execution time, and compiled code space usage for the existing system divided by that for the new system.

empty database and compiling each program. The rightmost bar is the total number of entries generated by compiling all four programs in succession, starting with an empty database. The numbers to the right of this bar indicate the number of new entries generated by each program in this successive compilation. Because many database entries are used by more than one of the programs, the total number of entries generated by compiling all four programs is much less than the sum of the number of entries generated by compiling each program starting with an empty database.



In our current database representation, each entry is approximately 150 bytes, making the cumulative database of 453 entries consume roughly 70 Kb. This space usage is nearly offset by the reduction in compiled code space for the four programs. The current representation of the database has not been tuned to minimize space requirements. By using denser encoding of the type group information and eliminating unnecessary pointer indirections, back-of-the-envelope calculations indicate that the space required for each entry can be halved. We plan to switch to a more compact representation of the database once the exact format of the database information has stabilized.

Although the storage required by the database seems relatively modest and is a small price to pay for the savings in compilation time, additional techniques could drastically reduce the number of entries in the database. One technique simply would discard any database entries which would lead the decision making process to elect to inline a method and which had not been heavily used. Because performing an inlining experiment that commits is only slightly more expensive than inlining without an experiment, performing repeated inlining experiments for these discarded entries would not hurt compilation time very much. Statically, 50% of the database entries leading to “yes” inlining decisions were reused less than three times for our large benchmarks, so discarding these entries would reduce database space costs to roughly 35 Kb (without the refinements of the database representation discussed above).

5 Future Work

The details of the decision making function that makes the final “yes” or “no” decision given the time and space cost/benefit information deserve further attention. One extension would support adaptive heuristics that would be more willing to trade space for time in program “hot spots” (heavily executed portions of the program). Dynamic profile information could help identify such hot spots.

When the compiler does not find an entry in the database for an inlinable method, it has the option of performing an experiment. In our current implementation, we elect to do an inlining experiment only if the existing source-level length heuristics would have decided to inline the routine. This means that our new system will not inline any method that is not inlined by the existing system. In the future, we would like to explore policies where the compiler chooses to inline experimentally a larger number of candidate methods, in hopes of increasing execution speed by finding new methods that would be beneficial to inline.

The current SELF compiler generates a customized version of a method for each receiver class used during program execution [Chambers *et al.* 89]. Customization lets the compiler determine statically the class of the receiver in the body of a customized method, enabling all sends to `self` to be inlined. In effect, each routine is partially evaluated with respect to class of the receiver. Receiver-based partial evaluation is a special case of a more general partial evaluation strategy, in which a specialized version of a routine could be compiled by customizing on any kind of static information available at a call site or dynamic information that can be quickly tested as part of the call. For example, a method could be customized on the type of important arguments, enabling the compiler to inline sends to these arguments as well. To be practical, this more general form of customization would have to be limited to those arguments which are most important for performance. Identifying these arguments then becomes a key component of a more aggressive optimizing implementation. Type group analysis might help identify important arguments; when computing type group information for an argument, the compiler identifies the amount of static type information used for optimizations, and the improvement to run-time performance attributable to those optimizations. We believe that the area of compiler self-monitoring offers new opportunities for adaptive, optimizing systems.

6 Related Work

Previous work on automatic inlining focuses primarily on attempting to maximize the direct benefits of inlining without too much increase in compiled code space [Scheifler 77, Allen & Johnson 88, Chang *et al.* 92]. In the context of this related work, indirect benefits of inlining tend to be relatively unimportant. Automatic inliners for languages like SELF have quite a different flavor, particularly because many things which would be built-in operators and control structures in other languages are entirely user-defined in SELF, and these user-defined routines need to be inlined aggressively to get good performance. Additionally, the indirect benefits of inlining often are more important in determining profitability than the simple direct costs.

Ruf and Weise describe a technique for avoiding redundant specialization in a partial evaluator for Scheme [Ruf & Weise 91]. When specializing a called routine for the static information available at a call site, their technique computes a generalization of the actual types that still leads to the same specialized version of the called routine. Other call sites with different static information can then share the specialized version of the called routine, as long as they satisfy the same generalization.

Our type group analysis computes similar generalizations of static argument types. Cooper, Hall, and Kennedy present a technique for identifying when creating multiple, specialized copies of a procedure can enable optimizations [Cooper *et al.* 92]. They apply this algorithm to the interprocedural constant propagation problem. Neither Ruf and Weise nor Cooper *et al.* provide a framework for evaluating the relative benefits of specific optimizations, however.

7 Conclusions

We have designed and implemented a technique that allows the costs and benefits of inlining to be accurately assessed at relatively low cost. By storing the results of inlining experiments in a persistent database, compilation of large benchmarks speeds up by almost a factor of two, with a corresponding loss of execution speed of less than 15%. Compared to the previous source-level length estimation strategy, the new experiment-based strategy is much more accurate at assessing the actual compiled code space costs and execution time gains of inlining a particular method at a particular call site. The new method also is less sensitive to superficial source-level changes that have little effect on the generated code, and consequently we hope it will adapt better as the source code for the SELF system evolves.

Good inlining decision making will become more important in the future, as new implementation techniques, such as more sophisticated type analysis, interprocedural type analysis [Agesen *et al.* 93], and adaptive recompilation systems [Hölzle *et al.* 91b], enable more methods to be inlined. With these increased opportunities for inlining come increased responsibility for inlining wisely.

The techniques of inlining experiments and maintaining an inlining database are not specific to SELF nor even to object-oriented languages. Any language which performs automatic inlining can benefit from more precise cost and benefit information to aid inlining decisions. We expect that our new approach would be most helpful for languages that move much of the basic functionality, such as control structures and the standard data structures, out of the language kernel and into user-level code.

Finally, the technique of type group analysis, used to identify how much of the available static type information was used to gain how much benefit in run-time performance, is more general than the specific application of making better inlining decisions. For example, it may form a key component of a system that automatically determines when it is profitable to produce customized versions of methods, improving upon the current decision to always customize on the type of the receiver and never customize on the types of the other arguments.

Acknowledgments

Ted Romer and Miles Ohlrich provided helpful comments on an earlier draft of this paper. This research has been supported by a National Science Foundation Research Initiation Award (contract number CCR-9210990), a University of Washington Graduate School Research Fund grant, and several gifts from Sun Microsystems, Inc.

References

[Agesen *et al.* 93] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type Inference of SELF. To appear in *ECOOP '93 Conference Proceedings*, Kaiserslautern, Germany, July, 1993.

[Aho *et al.* 86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[Allen & Johnson 88] Randy Allen and Steve Johnson. Compiling C for Vectorization, Parallelization, and Inline Expansion. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 241-249, Atlanta, GA, June, 1988. Published as *SIGPLAN Notices 23(7)*, July, 1988.

[Chambers *et al.* 89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Conference Proceedings*, pp. 49-70, New Orleans, LA, October, 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989. Also published in *Lisp and Symbolic Computation 4(3)*, Kluwer Academic Publishers, June, 1991.

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

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

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

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

[Dean & Chambers 93] Jeffrey Dean and Craig Chambers. Training Compilers for Better Inlining Decisions. Submitted to *OOPSLA '93*.

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

[Hölzle *et al.* 91a] Urs Hölzle, Bay-Wei Chang, Craig Chambers, Ole Ageson, and David Ungar. *The SELF Manual, Version 1.1*, unpublished manual, February, 1991.

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

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

[Scheifler 77] Robert W. Scheifler. An Analysis of Inline Substitution for a Structured Programming Language. In *Communications of the ACM 20(9)*, pp. 647-654, September, 1977.

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

Appendix A Raw Benchmark Data

All times are in milliseconds, and all space costs are in bytes. Each of the run-time numbers presented is an average of 10 runs of the program.

The raw data for compilations which used the original source-level heuristics is presented below:

Benchmark	Compile time	Run time	Code size
`parser`	43,073	357	113,232
`primMaker`	58,276	844	225,932
`pathCache`	13,720	2430	33,568
`deltaBlue`	11,995	1565	49,636

The data for compilations using the inlining database is given in the table below. We compiled and ran the benchmarks initially with an empty inlining database (presented in the columns labelled “Filling Database”), and then we compiled and ran them again using the filled database produced by the first compilation (presented in the columns labelled “Using filled database”).

	Filling database		Using filled database		
Benchmark	Compile time	Run time	Compile time	Run time	Code size
`parser`	32,624	386	22,016	384	80,288
`primMaker`	47,164	1146	39,635	1165	215,376
`pathCache`	13,407	2983	6,107	2903	24,212
`deltaBlue`	12,255	1475	11,312	1455	49,140

Statistics about the number of entries in and storage used by the database are presented below. The first four rows were computed by compiling each program, starting with an empty database. The last row gives the database statistics for compiling all four programs in succession, starting with an empty database.

Benchmark	# of entries	size
`parser`	318	52,440
`primMaker`	290	48,240
`pathCache`	156	27,468
`deltaBlue`	149	22,528
All 4 programs	453	70,128

The following table summarizes statistics on the frequency of reuse for database entries. The table should be interpreted as, for example, 124 entries were used only once (i.e. were never reused).

Number of Uses	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15+	Total
Number of entries	124	78	29	34	17	10	13	25	11	12	5	4	7	2	82	453