Towards Better Inlining Decisions Using Inlining Trials

Jeffrey Dean and Craig Chambers

Department of Computer Science and Engineering
University of Washington

Abstract

Inlining trials are a general mechanism for making better automatic decisions about whether a routine is profitable to inline. Unlike standard source-level inlining heuristics, an inlining trial captures the effects of optimizations applied to the body of the inlined routine when calculating the costs and benefits of inlining. The results of inlining trials are stored in a persistent database to be reused when making future inlining decisions at similar call sites. Type group analysis can determine the amount of available static information exploited during compilation, and the results of analyzing the compilation of an inlined routine help decide when a future call site would lead to substantially the same generated code as a given inlining trial. We have implemented inlining trials and type group analysis in an optimizing compiler for SELF, and by making wiser inlining decisions we were able to cut compilation time and compiled code space with virtually no loss of execution speed. We believe that inlining trials and type group analysis could be applied effectively to many high-level languages where procedural or functional abstraction is used heavily.

1 Introduction

Inlining is an important implementation technique for reducing the performance costs of language abstraction mechanisms. Inlining (also known as procedure integration and unfolding) not only confers the direct benefits of eliminating the procedure call and return sequences but also facilitates optimizing the body of the called routine in the context of the call site; sometimes these indirect post-inlining benefits dwarf the direct benefits. Inlining has long been applied to languages like C and Fortran, but it may be even more beneficial in the context of higher-level languages. Functional languages such as Scheme and ML [Rees & Clinger 86, Milner *et al.* 90], pure object-oriented languages such as Smalltalk and Eiffel [Goldberg & Robson 83, Meyer 92], and reflective systems such as CLOS and SchemeXerox [Bobrow *et al.* 88, Adams *et al.* 93] encourage programmers to write general, reusable routines and solve problems by composing existing functionality, leading to programs with very high call frequencies. Compilers and partial evaluators, such as Similix and Schism [Bondorf 91, Consel 90], can exploit inlining to reduce the cost of these abstraction mechanisms and thereby foster better programming styles.

Inlining is possible only when the compiler can determine statically the single target routine invoked by a call; in functional and object-oriented languages, this determination can require sophisticated analysis [Shivers 88, Hall & Kennedy 92, Chambers & Ungar 90, Palsberg & Schwartzbach 91]. But even if the call site is *potentially* inlinable, inlining may not be *profitable*. Care must be taken not to inline too much, or compilation time and compiled code could swell prohibitively. Inlining should only be applied where the benefits obtained by inlining outweigh the costs.

In many systems, the profitability of inlining a particular routine is hard-wired into the compiler. For example, the Smalltalk-80 compiler hard-wires the definition and optimized implementation of several basic functions from its standard library, and the Haskell standard prelude is fixed so that compilers can implement the functions in the standard library more efficiently [Hudak *et al.* 90]. A drawback of the hard-wiring approach is that built-in routines usually run much faster than user-defined routines, discouraging programmers from defining and using their own abstractions. Other systems, including C++, Modula-3, T Scheme, SchemeXerox, Common Lisp, Similix, and Schism [Stroustrup 91, Nelson 91, Slade 87, Adams *et al.* 93, Steele 90], allow programmers to indicate explicitly which routines are profitable to inline. While granting programmers fine control over the compilation process, this approach requires programmers to have a fair understanding of the language's implementation issues (an assumption becoming less likely as implementations become more sophisticated) and can be tedious if inlining must be applied heavily to get good performance. Additionally, most explicit declaration-based mechanisms do not allow programmers to specify that inlining is profitable only in certain contexts, or that inlining should only take place at particular high-frequency calls of some routine.

Our research investigates techniques for automatically deciding when inlining is profitable. Making good inlining decisions depends crucially on accurately assessing the costs and benefits of inlining. Previous automatic decision makers used simple techniques for estimating costs based on an examination of the target routine's source code (or unoptimized intermediate code), and consequently they failed to take into account the effect of post-inlining optimization of the target routine. Our work corrects this deficiency, leading to more accurate cost and benefit estimates and therefore better inlining decisions.

Our system assesses the costs and benefits of inlining by first experimentally inlining the target routine, in the process measuring the actual costs and benefits of that particular inline-expansion, and then amortizing the cost of the experiment (called an *inlining trial*) across future calls to that routine by storing the results of the trial in a persistent database. Because the indirect costs and benefits of inlining can depend greatly on the amount of the static information available at the call site (e.g., the static value or class of an argument), our system performs *type group analysis* to determine the amount of available call-site-specific static information that was exploited during optimization. Each database entry is guarded with type group information, restricting reuse of the information derived from an inlining trial to those call sites that would generate substantially the same compiled code.

We implemented and measured this approach in the context of an optimizing compiler for SELF [Ungar & Smith 87, Chambers & Ungar 91], a pure object-oriented language similar to Smalltalk but without any hard-wired operations or control structures. The SELF

To appear in 1994 ACM Conference on LISP and Functional Programming, Orlando, FL, June, 1994.

compiler exploits dynamic compilation, interleaving compilation with execution, to get fast turnaround times and to benefit from a form of profile information. By replacing the original heuristics for making automatic inlining decisions with an inlining trial-based approach, we sought to reduce compilation time while retaining the same level of run-time performance. Inlining trials were effective at this task: for four medium and two large SELF programs, compile time was reduced by an average of 20% with virtually no loss in run-time performance. We believe that in systems with more opportunities to inline than the optimizing SELF compiler we studied, inlining trials and type group analysis could make an even bigger improvement in the compile-time/run-time tradeoff.

The next section of the paper reviews previous techniques for making inlining decisions automatically. Section 3 describes inlining trials, with type group analysis detailed in section 4. In section 5 we present experimental measurements of our implementation. Section 6 describes some other related work, and section 7 concludes.

2 Previous Work on Automatic Decision Making

Existing compilers typically make automatic inlining decisions using an estimate of the cost of inlining based on an examination of the routine's unoptimized source code or intermediate representation. For example, the original SELF compiler counts the number of message sends in the candidate routine and inlines the routine if this number is below some threshold [Chambers 92]. The GNU `gcc` C compiler inlines a routine only if the number of instructions in its RTL (register transfer language) representation is less than some threshold [Stallman 90].

Source-level heuristics suffer from the problem that they do not consider the effect of optimizations applied to the body of the called routine after inlining, in particular those optimizations derived from static information available at the call site. For example, a hash table lookup routine may normally be considered too big to inline profitably. But if the key to the hash table is a compile-time constant, then some of the code of the lookup (such as computing the hash of the key) could be optimized away after inlining, making the lookup routine more attractive to inline. If the hash table itself is a compile-time constant, then the entire lookup routine can be constant folded away. Some partial evaluators can perform this sort of optimization already, but compilers typically are not tuned to inline so aggressively. Inlining trials allow the effects of post-inlining optimizations to be considered when making inlining decisions, and type group analysis allows call sites with differing amounts of available static information to be treated separately.

Source-level heuristics can be overly sensitive to the superficial form of the target routine. For example, the SELF compiler's original source-level heuristics had been tuned so that important routines such as the one implementing a `for`-loop were inlined. Several years later, the standard library was reorganized, and the definition of the `for`-loop routine was changed in a superficial way to be easier to read. The changed version appeared more complex to the compiler, however, and the compiler (silently) ceased to inline `for` loops. Performance on loop-intensive code mysteriously plummeted as a result. Such experiences, as well as only modestly-successful attempts to improve the source-level heuristics, provided the motivation for us to develop inlining trials. By assessing costs and benefits of inlining on the routine *after* optimization, inlining trials are much less sensitive to superficial details of the source code and can adapt as the source code evolves.

The Impact C compiler uses profile information to help guide the inlining process [Chang *et al.* 92]. The profile information is used to weight arcs in the program's call graph, allowing the cost/benefit estimates to be weighted by the expected execution frequency, and leading to better inlining decisions. Our current implementation of inlining trials does not incorporate profile data, instead relying on static estimates of execution frequency but profiling information would be easy to incorporate into an inlining trial-based system.

3 Inlining Trials

To make better inlining decisions, the compiler needs more accurate information on the actual costs and benefits of inlining a routine in the context of a particular call site. Accurate information can be obtained by tentatively inlining the routine, optimizing the inlined routine in the context of the call site, and then examining the resulting code. If the costs outweigh the benefits, the effects of inlining on the program representation can be undone. Such a conditional inline expansion, used to calculate the costs and benefits of inlining including the effects of optimization, we call an *inlining trial*.

Clearly, performing an inlining trial is much more time-consuming than estimating costs and benefits based on unoptimized source code. To regain acceptable compile-time costs, we save the results of each inlining trial in an *inlining database* that persists across compiles. Future opportunities to inline the same routine at other call sites consult the database instead of repeating the trial, thereby amortizing the cost of the trial over all uses of the information in the database. If a routine is called from many call sites, the amortized compile time cost of the trial can be small. Furthermore, if a few routines are identified that turn out to be bad choices to inline, the savings reaped by not inlining those routines can offset the cost of all the trials. Our experience using this approach in the SELF compiler is that many routines are invoked from multiple call sites; as reported in section 5, overall compilation time for an application actually *decreases* when using inlining trials.

The process involved in making an inlining decision is summarized by the following pseudocode:

F: the estimated execution frequency of the call site

R: the target routine

T: static information available at the call site

TG: type group information describing call site-specific static information exploited during inlining trial

c, *b*: cost and benefit information for inlining trial

D: inlining database = $R \times TG \rightarrow c \times b$

```
should-inline(R, T, F, D) =
    if ∃ (R, TG) ∈ dom(D) such that T ∈ TG then
        -- use database entry if available
        (c, b) ← D(R, TG)
    else if source-level-length(R) ≤ threshold then
        -- do inlining trial if simple source-level heuristic passes
        (c, b, TG) ← perform-trial(R, T)
        add ((R, TG) → (c, b)) to D
    else
        -- don't bother with trial
        (c, b) ← (∞, 0)
    end
    return make-decision(c, b, F)
```

The remainder of this section discusses inlining trials in more detail. Sections 3.1 and 3.2 describe how we estimate costs and benefits of inlining during a trial, respectively, and section 3.3 discusses how to make the final inlining decision given cost and benefit information. Section 3.4 addresses what happens when inlining is invoked recursively within a trial. Section 4 explains type group analysis, the mechanism whereby our system describes the amount of call-site-specific type information exploited when optimizing the inlined routine.

3.1 Estimating Costs

The major costs of inlining are increased compiled code size and increased compile times. Computing the space cost of inlining a routine is easy to measure: after optimizing the routine in the context of the call-site, the compiler sums the expected space needed to generate machine code for each control flow graph node in the body of the inlined routine; in our implementation this is an estimate since register allocation and instruction scheduling have not yet been performed. The compiled code space needed to generate a call is then subtracted from the space taken by the inlined routine to determine the total expected space cost for inlining.

Estimating the compile time required to inline a routine is more difficult. Simply using a timer to measure compilation time suffers from the low resolution of most hardware clocks. It also is difficult to calibrate across different compilation platforms and across versions of the compiler with differing levels of debugging instrumentation. Fortunately, compilation time in our system seems to be roughly proportional to compiled code space usage: we measured the compilation of 1,972 SELF procedures and found a correlation coefficient $r = 0.93$. Consequently, our implementation considers only compiled code space usage in the cost/benefit tradeoff.

3.2 Estimating Benefits

The major benefit of inlining we consider is reduced execution time through elimination of executed instructions. Time savings can be viewed either as an absolute savings or as a relative savings. Our implementation supports both views by computing two execution time estimates: the amount of time taken by an execution of the inlined routine and the number of instructions saved as a result of inlining, after optimizations have been applied. Absolute and relative estimated execution time savings can be calculated from these two numbers.

Computing an estimate of time taken in an invocation of the inlined routine requires estimating the time taken for each control flow graph node, after optimization, weighting it by its expected execution frequency, and summing. This calculation is mostly straightforward, using standard compiler static estimates for execution frequency. (Due to space constraints, some of the subtleties involved with this calculation are relegated to a separate technical report [Dean & Chambers 93].)

To determine the execution time saved as a result of inlining, the compiler monitors each optimization performed on the body of the inlined routine and estimates the number of dynamic machine instructions skipped as a result of the optimization, weighted by expected execution frequency. However, the compiler considers only those optimizations enabled by static information that was available at the call site; other optimizations would be performed whether or not the routine was inlined. During an inlining trial, the compiler maintains a data structure describing the subset of available static information derived from the call site. Only optimizations based on information in the subset affect the execution time saved as a result of inlining. The savings attributed to these optimizations, plus the direct savings of the eliminated call and return sequence, form the estimated savings in execution time due to inlining.

3.3 Making Final Inlining Decisions

Once the cost and benefit information for a call site has been obtained, either by performing an inlining trial or by locating an applicable entry in the database, the compiler must make a decision. This decision depends on the environment to determine the relative value of compile time, compiled code space, and execution time. Inlining trials provide better information upon which to base an inlining decision, but some controlling mechanism still needs to make a decision. For our implementation, we use a simple function that considers compiled code space cost and relative execution time savings and inlines the routine if the ratio of time savings to space cost is above a particular threshold; dynamic profile data could be include easily by weighting the expected execution time savings.

3.4 Nested Inlining

When optimizing an inlined routine, calls within the inlined routine may themselves be candidates for inlining. Optimizing these candidates can lead to recursive inlining trials. Such recursion poses no problems, and in fact occurs often in our implementation. The costs of inlining a routine include the costs associated with inlining any of its calls, and the benefits of inlining a routine include any benefits derived as part of inlining calls within the routine. The compiler must track the flow of static information from the outer call site through any contained calls, in order to correctly attribute the savings derived from some optimization to the appropriate source of static information.

4 Type Group Analysis

During an inlining trial, the compiler uses any information available statically at the call site to optimize the body of the inlined routine. Consequently, the costs and benefits of the trial reflect this call-site-specific information. For example, if at some call site the compiler knows the concrete type of an argument, accesses to the argument in the body of the inlined routine are likely to get optimized substantially, increasing the apparent benefits of inlining the routine. However, a different call site that lacked static information about the argument's type would be attributed a lower inlining benefit. If the results of an inlining trial for one of these two call sites were applied to the other, inappropriate inlining decisions might be made.

To avoid these potential problems, an inlining trial database entry is guarded with a description of the kind of static information that should be present at the candidate call site for the results of the trial to be reasonably predictive. During an inlining trial, the compiler monitors uses of static information derived from the caller and records the amount of static information that enabled (or disabled, in the case of a lack of static information) each optimization. This summary information is added to the inlining database entry storing the results of the trial. When a future call site searches the inlining database, the static information available at the call site must be compatible with the summary of an entry for it to match.

In our system, concrete type information about the arguments to the inlined call are the principal sources of optimization. Guarding an inlining trial's database entry with the actual concrete type information available about the arguments would be too specific, however: few call sites would have exactly the same static type information as the inlining trial, and consequently there would be little reuse of the results of inlining trials. Instead, the inlining entry should be guarded with a *description* of the kinds of static type information that lead to roughly the same degree of optimization. We call these descriptions of static type information *type groups*. A type group specifies a set of types, where all member types lead to substantially the same optimizations being performed as part of inlining. Types themselves in our system describe sets of values that share common properties relevant to the optimizations performed by the compiler. The main types represented in the SELF compiler are

the following (V is the set of all possible values, which is partitioned into a set of classes $\{C_1, ..., C_n\}$):

Type name	Set description	Meaning
UnknownType	V	Expressions of unknown concrete type
Class(C_i)	C_i	Instances of a class; most general type supporting inlining
Constant($v \in V$)	$\{ v \}$	Compile-time constant
Union($t_1, ..., t_n$)	$t_1 \cup ... \cup t_n$	Combined types, from merge CFG nodes
Difference(t_1, t_2)	$t_1 - t_2$	Certain types excluded, from failed value- and type-tests

In the same way that types represent sets of values, type groups represent sets of types. The following type groups are used in our extension of the SELF compiler (T stands for the set of all types):

Type Group name	Set description	Meaning
Universal	T	Any type
SubtypeGroup($s \in T$)	$\{ t \in T \mid t \subseteq s \}$	Any type which is at least as precise as s
AClass	$\{ t \in T \mid t \subseteq C, C$ a class type $\}$	Any type with class-level information
AClosure	$\{ t \in T \mid t$ is a closure type $\}$	Any closure type (a special kind of class info)
AConstant	$\{ t \in T \mid \|t\| = 1 \}$	Any type describing a compile-time constant
IntersectGroup($t_1, ..., t_n$)	$t_1 \cap ... \cap t_n$	Intersection of several type groups
ExcludeGroup($s \in T$)	$\{ t \in T \mid t \not\subset s \}$	Any type not in a certain type group

4.1 Using Type Group Information

Each argument of a database entry is guarded with a type group. For a database entry to be applicable to the call site, the static type information for each actual argument must be a member of the set specified by the corresponding type group. If for example the type group of some argument is the Universal type group, then any actual argument type will match; this implies that the optimization of the inlined routine does not depend on the static type information available for that argument. If instead the type group was IntersectGroup(SubtypeGroup(Fixnum), AConstant), then only actual arguments whose static type information conveyed that the argument was some fixnum constant would match. Such a precise type group implies that the compilation of the inlined routine is able to exploit the information that the argument is some fixnum constant, say through constant folding within the inlined routine, that would not be possible if less static information were available. As a final example, if the type group were ExcludeGroup(AClass), then only static types that were less specific than a concrete class type would match. Type groups that exclude the more precise kinds of type information ensure that inlining candidates do not match against database entries for trials that were unable to perform optimizations due to a lack of static information at the call site. In this specific example, the lack of class-level type information during the trial prevented some optimization, such as performing message lookup at compile-time or eliminating a run-time type check.

4.2 Computing Type Group Information

To compute type group information for each argument, the compiler performs *type group analysis*. Type group analysis is unusual in that it does not compute some abstraction of the values manipulated by the program being compiled, but rather it monitors the compilation process itself, computing how the compiler manipulates static type information. From this standpoint, type group analysis is a kind of *meta-analysis*.

Type group analysis is performed in parallel with regular concrete type analysis during an inlining trial. At the beginning of a trial, each argument to the inlined routine is associated with the Universal type group, indicating that, so far, no static information about the arguments has been used. Whenever an optimization is performed based on static type information derived from an argument, the type group associated with that argument is narrowed by intersecting it with a type group that represents the kind of static information that enabled the optimization. Similarly, whenever an optimization is *disabled* because of a lack of precision in the static type of an argument, the type group for that argument is intersected with an ExcludeGroup type group that rules out types that could have enabled the optimization. The following table indicates, for some of the more common optimizations performed in the SELF compiler, the type group intersected if the static information about the argument enabled or disabled the optimization:

optimization	if enabled	if disabled
perform message lookup at compile-time	SubtypeGroup(*the class*)	ExcludeGroup(AClass)
constant folding	AConstant	ExcludeGroup(AConstant)
eliminate fixnum, float, etc. type tests	SubtypeGroup(*the class*)	ExcludeGroup(AClass)
eliminate true, false value tests	AConstant	ExcludeGroup(AConstant)
inline-expand body of closure	AClosure	ExcludeGroup(AClosure)

The type groups calculated as part of type group analysis are intended to represent the largest set of argument types that would lead to the same optimizations being performed at a future call site. Further details of type group analysis and its implementation in the SELF compiler can be found in a separate technical report [Dean & Chambers 93].

4.3 An Example

We will use the following inlining candidate to illustrate how type analysis and type group analysis interact:

```
method growable_sequence::fetch(index) {
    if index < 0 or index > self.max_index then
        error("index out of bounds")
    endif
    return self.elems[index + self.base_index]
}
... seq.fetch(i) ...
```

Assume that the compiler knows the concrete class type of the `seq` variable statically and it has statically-bound the `seq.fetch` message to the `growable_sequence::fetch` method above. Consider the case the static type of the argument `i` is fixnum. The compiler consults the inlining database for a matching entry; assume this fails. Since the target method is not unreasonably large, the compiler begins an inlining trial. Initially the type group associated with the argument `index` is Universal; no optimizations yet exploit

any static information about `index`. The first operation within the routine sends the < message to `index`. The compiler examines the static type of `index`, discovers that `index` is a class type, and statically-binds and inline-expands the `fixnum::<` method (perhaps invoking a recursive inlining trial in the process). To reflect using class-level type information about the `index` argument, the compiler narrows the type group of `index` from Universal to IntersectGroup(Universal, SubtypeGroup(fixnum)), or simply SubtypeGroup(fixnum). The compiler also updates the benefit information for the trial to reflect saving more than a dozen cycles by eliminating the overhead of dynamic binding and the call/return sequences for the < message.

The compiler analyzes the body of the inlined < method. The built-in `fixnum::<` method first tests that its argument type is also a fixnum. It is, but since the argument to < is not being monitored as part of the inlining trial for `fetch`, no type group information is affected. After verifying that its arguments are fixnum's, the compiler attempts to constant-fold the comparison. This requires both arguments to be integer constants, which does not succeed. The compiler again narrows the type of `index` to indicate that its static type was not specific enough to enable the optimization, intersecting `index`'s type group with ExcludeGroup(AConstant) to give IntersectGroup(SubtypeGroup(fixnum), ExcludeGroup(AConstant)). This type group matches all types that are at least as specific as fixnum type but that are less specific than a fixnum constant. Such types include fixnum and Union(Constant(3), Constant(4)) but excludes Constant(17), UnknownType, and Union(fixnum, flonum). Note that the type group of `index` excludes types that are constants, but clearly it does not exclude integer values reaching that part of the program. Type group information can exclude overly specific *type* information, but the *values* described by the excluded types can still appear, as long as some more general type including the value is included in the type group.

The compiler visits each of the remaining operations in the inlined routine, but no additional narrowing of the type group of `index` occurs; additional time savings accrue, however, during optimization of the > and + messages. The compiler then completes the trial by creating a new database entry that records the compiled code size of the inlined `fetch` method, the expected cycle count of an execution of the inlined method, and the expected number of cycles saved as a result of inlining the `fetch` method. This entry is added to the database, guarded by the type group calculated for the `index` argument. Finally, the compiler makes a decision about whether the fetch method should be inlined, undoing the effects of the trial if not.

Subsequent statically-bound invocations of the `fetch` method examine this database entry. If their `index` argument type is at least a fixnum but not a fixnum constant, then the results of the database entry are consulted to determine whether inlining is warranted. If `index` is known statically to be a particular fixnum constant, then a new inlining trial is performed. During such a trial, the `index < 0` expression can be constant-folded, resulting in additional savings in execution time and compiled code space that might change the decision about whether the call site is profitable to inline. Similarly, if the static type of the argument is less specific than a fixnum, or is some other class type, then a new inlining trial is performed to assess the costs and benefits of a different kind of static type information about `index`.

Without some mechanism like inlining trials and type groups, the compiler could examine only the unoptimized source code for the `fetch` method. In this and many similar cases, the kind of static information about the arguments to the call can have a significant effect on the nature of the final code; some calls will be profitable to inline, while others will not be. Inlining trials provide the compiler with more accurate information upon which to make decisions, and type groups enable the compiler to distinguish among call sites with different available static information.

5 Experimental Results

Our original motivation for developing the technology of inlining trials was to improve the response time of the optimizing SELF compiler. In the SELF system, compilation is interleaved with program execution, and a slow compiler leads to slowly running programs. Consequently, we attempted to construct an inlining decision maker that would lead to a significant decrease in compilation time without a major loss in execution speed; other environments might choose different tradeoffs, such as improving execution speed without a major loss of compilation speed.

To assess the effectiveness of inlining trials, we compared our new inlining decision making system using inlining trials against the source-level heuristics found in the existing SELF compiler, measuring compilation time, execution time, and compiled code space consumption. To make a more direct comparison, we set the initial "reasonably short" threshold (identifying routines where performing an inlining trial seems feasible) to exactly the same value used by the source-level heuristics. Thus the only difference between the two decision makers is that the new system might choose not to inline something that the existing system would inline. We examined the following suite of programs:[*]

Program	Source (lines)	Description
parser	400	Parser for an old version of SELF
primMaker	1,300	Program to generate wrapper functions from an interface description file
pathCache	300	Traverses the SELF object graph and assigns path names to objects
deltaBlue	600	Incremental constraint solving program
cecilInterp	10,700	Interpreter for the Cecil language
cecilCompiler	12,500	Compiler for the Cecil language

We suspected that the existing heuristics, tuned initially on small benchmarks, over-inlined for these larger programs. This over-inlining led to slower compiles and more space-consuming compiled code without much benefit in execution speed. We hoped that inlining trials would make better decisions on what routines were profitable to inline. Figure 1 on the following page reports the compilation time, execution time, and code space usage of these six programs for our new system, relative to the existing system.[†] (Appendix A includes the raw data.) Shorter bars indicate better performance for the new system. The chart shows compilation times both for starting with an empty inlining database for each program ("cold") and for starting with a filled inlining database ("warm"). The warm compiles were measured by reusing the database generated during the "cold compile" for the benchmark. In practice, since the database is persistent and entries are shared across programs, the compilation

[*] We also examined a large number of small benchmarks, used during the original development of the SELF compiler. The inlining trial-based system achieved the same compile-time and run-time performance as the existing system, as we hoped.

[†] The values in the chart are calculated as the compilation time, execution time, and compiled code space usage for the new system divided by that for the existing system, converted to a percentage. Execution time denotes just the time spent executing compiled code, not the time spent compiling the code.

Figure 1: Ratio of Inlining Trials to Old Heuristics

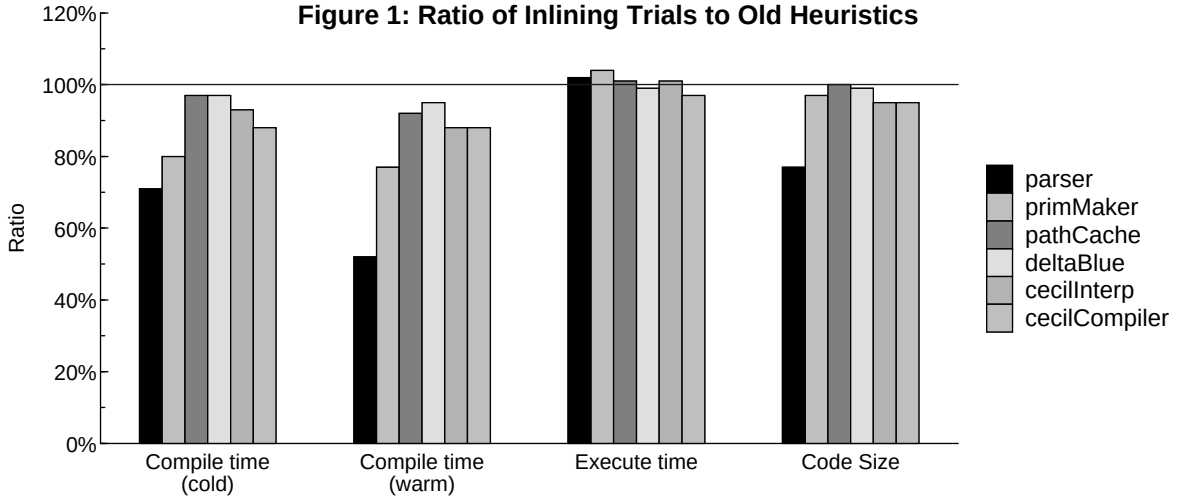


Figure 2: Compilation and Execution Time for a Range of Thresholds

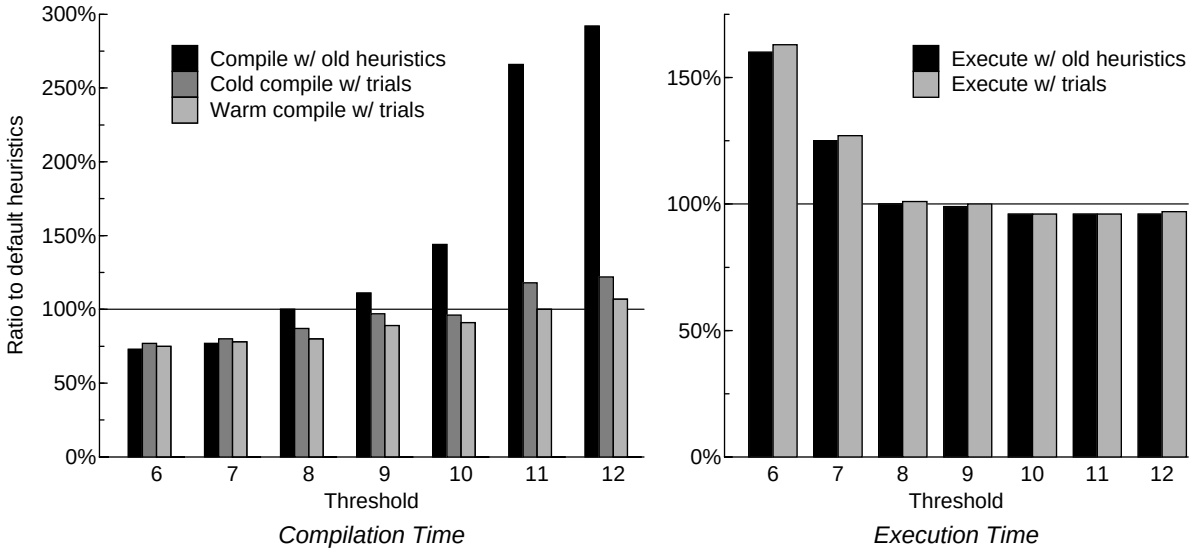
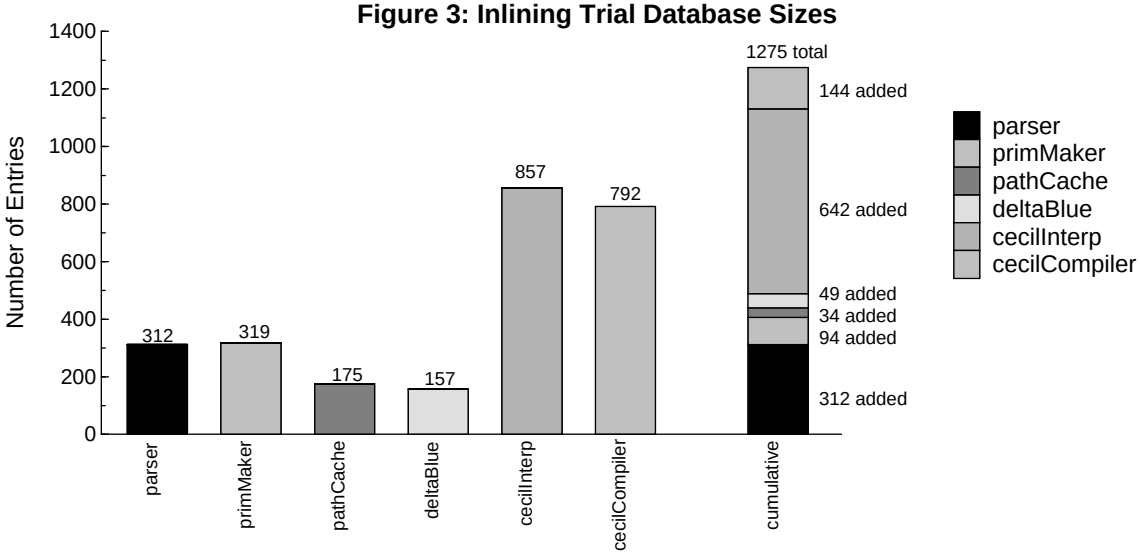


Figure 3: Inlining Trial Database Sizes



performance is closer to the warm compile figures than to the cold compile figures.

On average (using geometric mean), compilation time decreases by 20%, execution time increases by an average of 1%, and code space usage decreases by 6%. On an absolute scale, the compilation time savings of 20% represents a savings of 68 seconds of the 291 seconds required to compile all six programs; in our environment, compilation time is a significant cost worthy of optimization effort. Based on these results, we consider inlining trials to be effective at meeting our goal of balanced compilation and execution times.

The parser program shows particularly good improvement in compilation time. Under the old source-level heuristics, the `advance` routine, called to move the current character position forward in the input buffer, was inlined 26 separate times. However, `advance` does not benefit much from type information available at the call site, so there is little indirect benefit to inlining. The inlining trial-based system detected this and consequently never inlined the `advance` routine, saving a lot of compilation time and code space in the process.

The compilation improvement shown by these programs, while quite significant, is not as impressive as it might be in another environment. For these programs, the current SELF compiler is unable to statically bind many messages because of a lack of static type information. Future compilers for SELF and other object-oriented languages [Chien *et al.* 93, Hölzle & Ungar 93, Chambers *et al.* 93] are expected to incorporate interprocedural type analysis and extract type information from execution profiles, leading to many more messages being statically bound and thus eligible for inlining. We expect the importance of making good inlining decisions to grow as other parts of the compiler become more effective.

The above experiments used the same initial threshold for both systems. To see how sensitive the two approaches are to the choice of this threshold, we repeated the comparison of the two systems on the large programs for a range of thresholds. In figure 2, on the previous page, we report the geometric mean of compilation time and execution time for the two systems on the six-program benchmark suite for several different thresholds. The values in the chart have been normalized to the performance of the old inlining heuristics when using the default threshold of 8. Increasing the threshold value increases the number of routines considered for inlining.*

Compilation time is much less sensitive to the choice of threshold under the new inlining trial-based heuristics than under the old source-length heuristics, and the new approach has significantly better compilation time behavior than the old system. Also, the new inlining trial-based decision-making achieves nearly the same execution speed as under the old heuristics. Together, these results illustrate some of the different compile-time/run-time tradeoffs that can be made. In our system we set the threshold to 8, leading to a 20% reduction in compilation time with a negligible loss of execution speed. If instead we set the threshold to 10, compilation time would still drop by 9% but run time would also drop by 4%. Because compilation speed does not degrade much when using a higher initial threshold under the new system, we can use a higher threshold and be more robust in the face of future superficial changes to the source code of libraries and applications, such as the superficial rewrite of the for-loop implementation described in section 2.

Figure 3 on the previous page reports the number of database entries generated by compiling the large programs (Appendix A includes the raw data). The first six columns represent the number of entries created when compiling each program individually against an initially empty database. In our implementation, each database entry takes up approximately 75 bytes of space; the savings in compiled code space for using inlining trials compensates for the additional space cost of the database entries, and the compiled code space savings persist after program development ceases. The rightmost column indicates the total number of entries generated by compiling the six programs in succession, starting with an initially empty database. The numbers to the right of this column 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, such as entries for functions in the standard library, the total number of entries generated by compiling all six programs in succession (1275) is only half of the sum of the number of entries generated by compiling each program separately (2612).

6 Related Work

Previous work on automatic inlining has focused 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 higher-level functional and object-oriented languages have quite a different flavor, particularly because many things which would be built-in operators and control structures in lower-level languages tend to be user-defined in higher-level languages, and these user-defined routines need to be inlined aggressively to get good performance. Additionally, in the context of higher-level languages, 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, Ruf & Weise 92]. When specializing a called routine using 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 summary information about argument types, although the details of the two analyses differ.

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. To reduce the number of specialized copies of a procedure, their system evaluates when merging two specialized versions of a procedure would not sacrifice an important optimization. Our type group guards on database entries accomplish a similar task, enabling the results of an inlining trial to be reused for those call sites where similar optimizations are enabled, but over a richer domain of types.

* The existing source-level length heuristic is computed by summing weighted values for non-trivial message sends within the target routine. Certain messages which the compiler expects to be optimized (such as "+" and "`at:`") are assigned a weight of 1 and other message sends are assigned a weight of 2. A routine is eligible for inlining if the weighted sum of its messages is less than or equal to the inlining threshold.

7 Conclusion

Inlining trials are a promising mechanism for gathering more accurate information about the costs and benefits of inlining in an optimizing compiler. Better information can in turn lead to better automatic decisions about which call sites to inline. If these automatic decisions are good enough, standard library routines won't need to be hard-wired into the compiler for performance and programmers won't need to annotate routines with explicit inline directives. Ultimately, good automatic inlining can foster a better programming style by making the use of abstraction cheaper.

Unlike standard source-level inlining heuristics, inlining trials can consider the effect of post-inlining optimizations when assessing the costs and benefits of inlining. This provides the compiler with more accurate data upon which to base its inlining decision, and the post-optimization data is much less sensitive to superficial details of the source code. By storing the results of trials in a persistent database, the extra cost of a trial can be amortized across uses of the information. Type group analysis is key to reusing database entries for exactly those call sites whose static information would lead to the same set of optimizations being performed. Type group analysis may be applicable to other compilation problems, such as deciding when procedure specialization is profitable.

We have applied the language-independent ideas of inlining trials and type group analysis to improving the response time of the optimizing SELF compiler. In our implementation, the use of inlining trials cut compile time by 20% with virtually no effect on execution speed. By changing the cost/benefit tradeoff embodied by the final inlining decision-maker, we could have saved both compile time *and* execution time by making more intelligent inlining decisions. The extra compile-time cost of inlining trials is more than paid for by avoiding over-inlining. Incorporating dynamic profile information could improve the results even more.

Inlining trials and type group analysis appear most useful for languages where procedural abstraction is used heavily, where the compiler can determine statically the single target of a call, and where the effects of post-inlining optimizations are substantial and can vary across call sites. Many high-level functional and object-oriented languages meet this description. As the analyses of the targets of call sites improve, the compiler will have more opportunities to inline and consequently bear more responsibility for making wise decisions.

Acknowledgments

Susan Eggers, David Notkin, Erik Ruf, and Daniel Weise 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

[Adams *et al.* 93] Norman Adams, Pavel Curtis, and Mike Spreitzer. First-Class Data-Type Representations in SchemeXerox. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pp. 139-146, Albuquerque, NM, June, 1993. Published as *SIGPLAN Notices 28(6)*, June, 1993.

[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.

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

[Bondorf 91] Anders Bondorf. *Similix Manual, System Version 4.0*. Technical report, DIKU, University of Copenhagen, Copenhagen, Denmark, 1991.

[Chambers & Ungar 90] 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*, pp. 150-164, White Plains, NY, June, 1990. Published as *SIGPLAN Notices 25(6)*, June, 1990. 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.

[Chambers *et al.* 93] Craig Chambers, Jeffrey Dean, Dave Grove, and Charlie Garrett. Analysis and Optimization of Object-Oriented Languages. Unpublished manuscript, October, 1993.

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

[Chien *et al.* 93] Andrew A. Chien, Vijay Karamcheti, John Plevyak. The Concert System: Compiler and Runtime Support for Efficient, Fine-Grained Concurrent Object-Oriented Programs. Technical report R-93-1815, Department of Computer Science, University of Illinois at Urbana-Champaign, 1993.

[Consel 90] Charles Consel. *The Schism Manual, Version 1.0*. Yale University, New Haven, CT, December, 1990.

[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 to Make Better Inlining Decisions. Technical report 93-05-05. Department of Computer Science & Engineering, University of Washington, Seattle, WA, May, 1993.

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

[Hall & Kennedy 92] Mary W. Hall and Ken Kennedy. Efficient Call Graph Analysis. In *ACM Letters on Programming Languages and Systems 1(3)*, pp. 227-242, September, 1992.

[Hölzle & Ungar 93] Urs Hölzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. Unpublished manuscript, 1993.

[Hudak *et al.* 92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. *Report on the Programming Language Haskell, Version 1.2*. In *SIGPLAN Notices 27(5)*, May, 1992.

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

[Milner *et al.* 90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

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

[Palsberg & Schwartzbach 91] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *OOPSLA '91 Conference Proceedings*, pp. 146-161, Phoenix, AZ, October, 1991. Published as *SIGPLAN Notices 26(10)*, October, 1991.

[Rees & Clinger 86] Jonathan Rees and William Clinger, editors. *Revised^3 Report on the Algorithmic Language Scheme*. Published as *SIGPLAN Notices 21(12)*, December, 1986.

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

[Ruf & Weise 92] Erik Ruf and Daniel Weise. Avoiding Redundant Specialization During Partial Evaluation. Technical Report 92-518. Department of Computer Science, Stanford University, Stanford, CA, 1992.

[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.

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

[Slade 87] Stephen Slade. *The T Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[Stallman 90] Richard M. Stallman. *Using and Porting GNU gcc Version 2.0*. Free Software Foundation, November, 1990.

[Steele 90] Guy L. Steele, Jr. *Common Lisp: The Language, second edition*. Digital Press, Bedford, MA, 1990.

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

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

Appendix A Raw Data

The following table shows the raw data for the experiments. All times are in milliseconds, and all ratios are relative to the compilation and execution times for the old heuristics with the default threshold of 8 (shown in bold):

		Threshold	6		7		8		9		10		11		12	
		Program	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
Compile	Compile - old heuristics	parser	13278	0.40	15836	0.48	**32834**	**1.00**	35322	1.08	42748	1.30	87252	2.66	92358	2.81
		primitiveMaker	33773	0.71	34849	0.73	**47666**	**1.00**	49218	1.03	68345	1.43	250143	5.25	256611	5.38
		pathCache	5021	0.88	4922	0.87	**5683**	**1.00**	6816	1.20	7108	1.25	7209	1.27	7700	1.35
		deltaBlue	9216	0.92	9148	0.92	**9969**	**1.00**	11821	1.19	11784	1.18	12166	1.22	12253	1.23
		cecilInterp	79425	0.82	87456	0.90	**96895**	**1.00**	104681	1.08	157452	1.62	306041	3.16	363681	3.75
		cecilCompiler	78919	0.80	84398	0.85	**98719**	**1.00**	105818	1.07	196308	1.99	507091	5.14	536549	5.44
		Geometric mean		0.73		0.77		**1.00**		1.11		1.44		2.66		2.83
	Compile w/trials - cold	parser	14327	0.44	17518	0.53	23304	0.71	25212	0.77	26014	0.79	36510	1.11	38554	1.17
		primitiveMaker	33836	0.71	34613	0.73	37942	0.80	42553	0.89	43948	0.92	78707	1.65	81599	1.71
		pathCache	5401	0.95	5374	0.95	5488	0.97	6062	1.07	5949	1.05	6125	1.08	6291	1.11
		deltaBlue	9454	0.95	9463	0.95	9706	0.97	10459	1.05	9746	0.98	10303	1.03	10361	1.04
		cecilInterp	85486	0.88	85721	0.88	89775	0.93	104063	1.07	102252	1.06	122176	1.26	128032	1.32
		cecilCompiler	80398	0.81	83439	0.85	86947	0.88	97378	0.99	97302	0.99	103427	1.05	104016	1.05
		Geometric mean		0.77		0.80		0.87		0.97		0.96		1.18		1.22
	Compile w/ trials - warm	parser	13444	0.41	15080	0.46	17042	0.52	21488	0.65	20785	0.63	24530	0.75	27405	0.83
		primitiveMaker	33549	0.70	35000	0.73	36805	0.77	41529	0.87	42988	0.90	58341	1.22	76548	1.61
		pathCache	5148	0.91	5248	0.92	5227	0.92	5693	1.00	5897	1.04	5652	0.99	5823	1.02
		deltaBlue	9362	0.94	9484	0.95	9514	0.95	9591	0.96	9680	0.97	9669	0.97	9712	0.97
		cecilInterp	82777	0.85	84771	0.87	85181	0.88	93261	0.96	95259	0.98	107946	1.11	105125	1.08
		cecilCompiler	81034	0.82	82892	0.84	86749	0.88	91747	0.93	97294	0.99	102350	1.04	102949	1.04
		Geometric mean		0.75		0.78		0.80		0.89		0.91		1.00		1.07
Execute	Execute - old heuristics	parser	972	2.14	764	1.68	**455**	**1.00**	446	0.98	412	0.91	428	0.94	432	0.95
		primitiveMaker	1877	1.80	1501	1.44	**1042**	**1.00**	1030	0.99	1001	0.96	1037	1.00	949	0.91
		pathCache	3019	1.33	2418	1.06	**2274**	**1.00**	2274	1.00	2282	1.00	2286	1.01	2130	0.94
		deltaBlue	2364	1.70	1522	1.09	**1391**	**1.00**	1329	0.96	1247	0.90	1254	0.90	1257	0.90
		cecilInterp	32475	1.11	29124	0.99	**29293**	**1.00**	30492	1.04	29452	1.01	29169	1.00	32059	1.09
		cecilCompiler	2871	1.74	2232	1.35	**1649**	**1.00**	1642	1.00	1628	0.99	1559	0.95	1621	0.98
		Geometric mean		1.60		1.25		**1.00**		0.99		0.96		0.96		0.96
	Execute - trials	parser	963	2.12	750	1.65	466	1.02	466	1.02	446	0.98	430	0.95	425	0.93
		primitiveMaker	1891	1.81	1540	1.48	1088	1.04	1070	1.03	1032	0.99	1053	1.01	1023	0.98
		pathCache	3129	1.38	2406	1.06	2293	1.01	2308	1.01	2191	0.96	2225	0.98	2229	0.98
		deltaBlue	2409	1.73	1567	1.13	1381	0.99	1348	0.97	1274	0.92	1257	0.90	1269	0.91
		cecilInterp	32853	1.12	30649	1.05	29487	1.01	29929	1.02	28907	0.99	29104	0.99	30745	1.05
		cecilCompiler	2971	1.80	2228	1.35	1597	0.97	1593	0.97	1556	0.94	1560	0.95	1640	0.99
		Geometric mean		1.63		1.27		1.01		1.00		0.96		0.96		0.97

The following table shows code sizes for the six programs compiled with the old heuristics and with inlining trials (with a threshold of 8):

Program	Code size-old		Code size - trials	
parser	107676	1.00	83040	0.77
primitiveMaker	233148	1.00	225812	0.97
pathCache	30164	1.00	30160	1.00
deltaBlue	51036	1.00	30616	0.99
cecilInterp	421400	1.00	399428	0.95
cecilCompiler	429948	1.00	407080	0.95
Total	1273372	1.00	1196136	0.94