Method Inlining in the Titanium Compiler

CS265 Semester Project Report

Dan Bonachea, *bonachea@cs.berkeley.edu*

Abstract

Titanium is a parallel dialect of Java designed for writing high-performance scientific applications. In this paper, we discuss the design and implementation of a method inlining optimization added to the Titanium compiler. We address issues relevant to the design of any inlining optimization, and specifics relevant to Java and Titanium. Two heuristics are developed for guiding the inlining decisions, a manual, programmer-assisted heuristic and an automatic heuristic based on call graph structure and an analysis of method characteristics. Results are presented detailing the effectiveness of each heuristic and the performance improvements realized.

1 Introduction

Method inlining is an important and well-studied interprocedural optimization, where a method call is replaced by a copy of the called method body, and the code is fixed up appropriately to preserve the semantics of the call. This transformation is often beneficial for two reasons: first, it removes the direct cost of the method call and dispatch, which can be quite significant on modern, deeply-pipelined processors. Secondly, (and perhaps more importantly) the inlining operation indirectly benefits subsequent optimizations - replacement of the opaque method call with equivalent code can make it easier for intraprocedural optimizations (such as copy propagation, loop optimizations, common subexpression elimination, etc.) to analyze the enclosing method and therefore optimize it more aggressively. It is also frequently the case that the inlined code can be specialized significantly thanks to the information available in the caller method's context about the parameter values (for example, inlining a call site that has some constant actual parameters often allows dead-code elimination to remove entire basic blocks from the inlined callee body).

The most significant downside to method inlining is that each inlining operation essentially adds a new copy of the callee body to the program, which implies a subsequent increase in the size of the executable image. The technical term for this problem is "code bloat". Code bloat can lead to a larger runtime memory footprint and decreased I-cache performance, which generally imply a runtime performance penalty. Code bloat also leads to increased compilation/optimization time and memory overhead. Clearly we must seek to strike a balance between the benefits and potential costs of inlining.

2 Background - Titanium compilation model

Some basic information about the Titanium compilation model is necessary to understand the design and workings of the method inliner, since it often performs inlining operations that cross module and library boundaries.

A Titanium executable is comprised of 4 primary source components:

- User Titanium source – this is the user's actual program source (backend-independent)
- Library Titanium source – the java.* and ti.* standard library hierarchy, also known as the tlib (backend-independent)
- Native code – the hand-written C implementation of the native methods declared in Titanium source
- Runtime library – a backend-specific library written in C which provides low-level functionality such as network messaging, garbage collection and the native implementation of language primitives such as barrier, broadcast and monitors.

The primary function of the Titanium compiler (tc) is to read and parse the Titanium source that comprises the first 2 components and output corresponding C code which is written to a temporary directory. A standard ANSI C compiler (such as gcc) is used to compile this generated C code and the hand-written C code comprising the last 2 categories, and everything is linked together into the final executable using the standard UNIX linker. However, the order in which these steps happen can vary somewhat based on the compilation options supplied by the user.

In the simplest compilation scenario (using the --rebuild-tlib option, and hereafter referred to as a "rebuild-tlib" compilation), both the user and library Titanium source are compiled and written to C by tc, and then compiled and linked with the pre-existing runtime library (which is built at compiler installation time). One advantage to this model is that it offers true full-program compilation, which simplifies some interprocedural analyses and optimizations (such as local and sharing qualification inference). However, this compilation scenario can result in very long build times (up to an hour on some systems!), because the tlib is very large and the resulting generated C code takes a long time to process (in actuality, tc only generates code for classes actually referenced by the user program and certain crucial system classes, but in practice this amounts to a large fraction of the library).

To combat this performance problem, a second option is offered (hereafter referred to as a "standard" compilation) whereby tc only generates C code for the user Titanium source, and (once compiled) these modules are linked to a pre-compiled version of the library Titanium source[1]. This option offers build time roughly linear in user program size, but presents some challenges to interprocedural analyses and optimizations, such as method inlining.

See the Titanium web site [17] for more information about Titanium.

[1] There is actually a third distinct mode of compilation we will call "library-only", which is used to build the pre-compiled version of the tlib that is linked to the user application in a "standard" build. The distinguishing qualities of interest for library-only builds is that it is a partial-program compilation (as the user source is not available at that time), and no main() method is available.

3 Method Inlining

In this project, we added a method inlining optimization to the Titanium compiler.

There were several issues to be addressed in designing an inliner. For example, which call sites are candidates for inlining? How do we decide which of the candidates to inline? Finally, how is the inlining operation itself performed? We now address each of these issues in turn.

4 Call Site Eligibility for Inlining

An important question to be answered in the design of any inliner is which method calls are candidates for inlining – there several factors which can make it impossible or inadvisable to inline certain call sites.

The most obvious restriction is that the method call must be statically bound – that is, we must be able to uniquely identify the correct callee of a call site at compile time, and furthermore the callee body must be available to inliner. Object-oriented languages such as Titanium support virtual method dispatch, where the method callee is based on the dynamic type of the target object, which in general may be any subtype of the static type visible at compile-time. In the absence of other information or assumptions about the dynamic type, it is only safe to perform the inlining operation on a virtual call site if it can be determined that only one possible callee exists.[2]

```
class A {
  public int foo() { … }
}
class B extends A {
  public B() { … }
  public int foo() { … }
  public int bar() { … super.foo(); … }
}
…
A a = new B();
…
a.foo();
```

For example, in the code above, the call `a.foo()`cannot be inlined because the static type of the target object (a) doesn't give us enough information to uniquely determine which method override should be called. There may be some cases where the dynamic type can be inferred using a dataflow analysis (Ghemawat, Randall and Scales [10] discuss one such analysis), but in general this problem is undecideable and beyond the scope of this research.

[2] Some research has been done on techniques whereby the optimizer inserts checks for the dynamic type of an object or set of objects, and branches to a version of the given code optimized using corresponding assumptions about the dynamic types (see [3], [9] for details) - however, we won't consider that approach any further in this paper.

Because the majority of method calls in Java are virtual calls, and the language provides dynamic class loading (which allows an arbitrary, previously unseen class to be loaded at runtime), inlining general method calls can be very tricky – at any time during the program execution a new subclass may be loaded which overrides supertype methods and potentially invalidates previously inlined calls to those methods. Significant research has gone into techniques for effective inlining in Java using a runtime optimizer (JIT optimization). See [12] for further details.

Although Titanium supports most of the features of Java 1.0, it does not allow dynamic class loading – primarily to provide a simple solution to the difficulties described above and because that feature was deemed of little use to high-performance scientific applications. The Titanium compilation model is also generally a full-program compilation (as opposed to separate compilation which makes compile-time inlining unsafe in Java), which means that while compiling a user's application, all possible method targets are visible to the inliner and we can safely decide if a virtual call has a unique target (making it a candidate for inlining)[3].

Luckily, there are several important special cases where method calls can always be statically bound. Most notably, calls to constructors are always nonvirtual, which is useful because constructor bodies are often trivial and make good candidates for inlining. Other special cases that are always statically bound include calls of the form "super.methodname", calls to methods that are private, static or final and calls to methods in final classes, which cannot be overridden. The last is especially important in Titanium because of immutable classes, which are implicitly final (see the Titanium language reference [11] for details on immutable classes) – calls to methods in immutable classes are always nonvirtual.

An important case to be considered in designing an inliner is method calls which are recursive or lie along a mutually recursive call cycle. Failure to properly detect and handle such a situation can easily lead to non-termination of the inlining phase. The detection of mutually recursive call cycles requires the construction of an interprocedural call graph. However (as suggested by Kaser et al. in [15]) for the vast majority of programs it is sufficient to forbid inlining of calls to directly recursive methods to ensure termination (most mutually recursive cycles reduce to direct recursion after some finite number of inlining steps). Note that this is NOT the same as simply forbidding inlining on directly recursive calls – this superficially similar policy is often insufficient to prevent nontermination (for example, inlining a call to a directly recursive method can lead to a functionally identical call graph, implying the possibility of an infinite number of such steps).

[3] An exception to this rule occurs during "library-only" compilations that do not offer full-program analysis – in that case we must conservatively assume that unseen overrides may exist for methods wherever modifiers do not forbid them.

Yu and Kaser [19] note that even the policy of prohibiting inlining of calls to directly recursive methods is insufficient to prevent non-termination of the inliner on certain contrived programs. For example, in the call graph shown in figure 4.1 (where we assume each edge corresponds to a single call site), inlining the call from A to B leads to the call graph shown in figure 4.2, and subsequently inlining the call from A to C leads back to the call graph shown in figure 4.1, implying a potentially infinite oscillation, even though no direct recursion is present.

Figure 4.1

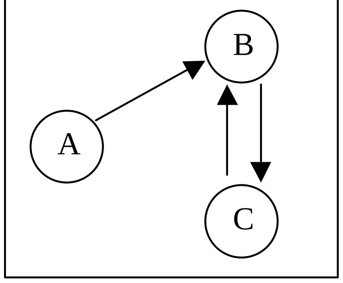
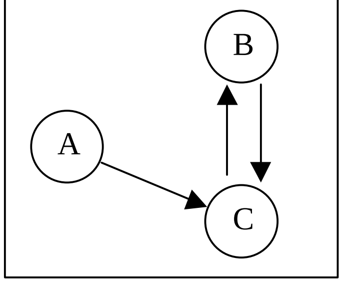


Figure 4.2



Clearly some additional restrictions are required to prevent non-termination on these rare (or devious) examples. In the next section we'll discuss the heuristics used to decide which candidate call sites should actually be inlined, but to guard against non-termination of the above variety we impose an arbitrary limit on the maximum depth of the inlining operations on any given call site (we found that 4 is a reasonable maximum depth and is almost never overly restrictive in real programs)[4].

Finally, our inliner chooses to never inline calls to native methods for practical reasons – see section 10.2 for more discussion.

5 Inlining Heuristics

In practice, a significant fraction of the method call sites in real programs are candidates for inlining (that is, they satisfy all the restrictions on which sites can be legally inlined, as discussed in section 4). However, choosing to inline all possible sites would lead to an exponential explosion of code bloat – such uncontrolled bloat would most likely bring the optimizer to a screeching halt, or at best lead to a very large, slow-running executable.

Clearly the inliner must make some choices about which sites to inline and which to leave unchanged. With the exception of a few rare domain-specific studies (e.g. Leupers & Marwedel [16]), most research seems to agree that finding a truly optimal solution to this decision problem is not tractable, or at least not worthwhile. Instead, most inliners use some sort of heuristic to drive the call site selection process. These heuristics generally seek to minimize dynamic call frequency and maximize subsequent

[4] Note this discussion assumes that call sites contained in the inlined callee body are candidates for subsequent inlining steps, as is the case in our inliner. If such sites are not considered for subsequent inlining, then this problem does not arise.

optimization opportunities created by inlining while keeping code bloat to a manageable minimum.

Kaser and Ramakrishnan [14] present an interesting and useful framework for thinking and reasoning about method inlining. They separate the largely orthogonal concepts of an inlining policy (the restrictions on which methods can be inlined, for example “only leaf methods”) and the inlining strategy (the algorithm used to traverse call sites and apply the inlining policy in order to reach some goal, such as minimizing runtime method calls).

Following their example, the design of our inliner is roughly divided into an inlining policy and strategy. We developed two different inlining heuristics – the “manual” heuristic has a primitive strategy and policy, relying primarily on programmer annotations to drive the inlining decisions. The “automatic” heuristic adds more sophisticated decision-making capabilities based on a global call-graph analysis.

5.1 Manual Inlining Heuristic

The Titanium language adds the new keyword “inline” to the list of standard method declaration modifiers defined by Java. The keyword has no language-level semantics, but is meant to allow the programmer to provide a “hint” to the inliner that calls to the method in question should be inlined whenever possible.

The manual heuristic has a very primitive inlining policy that relies primarily on the programmer annotating certain methods with the “inline” keyword - all candidate sites whose callees are marked as “inline” by the user are selected for inlining. The only such sites which are not inlined are those which fail to meet the eligibility requirements detailed in section 4 (e.g. calls to directly recursive methods or those with an uncertain callee). In addition, a short hard-coded list of methods are assumed to implicitly carry the “inline” annotation for the purposes of inlining decisions: the trivial java.lang.Object.Object() constructor (which is called during the creation of any object), all of the methods in java.lang.Math (e.g. Math.abs()), and any compiler-generated zero-argument constructors (empty constructors which are automatically generated for classes lacking a constructor).

The inlining strategy for the manual heuristic is a very simple one – all method definitions that are selected for code generation are processed one-at-a-time in random order, and each call site is processed in textual order by the inlining policy. Call sites inlined into the caller are also considered for subsequent inlining operations. All inlining operations use current-version inlining. No call graph is built, and attribute information needed by the policy (e.g. whether or not a method is directly recursive) is computed and updated lazily as necessary.

5.2 Automatic Inlining Heuristic

The manual inlining heuristic is a useful tool, but its effectiveness is somewhat limited for several reasons. The use of inlining annotations requires a good deal of thought and cleverness on the part of the programmer in order to strike a reasonable balance between call reduction and code bloat, and it is unclear how well-equipped the programmer is to make the right decisions. In addition, the inline annotation is rather constrained in its expressiveness, because it only allows the programmer to annotate at method declaration sites – however, the inlining decisions made by the optimizer are more directly related to particular call sites. One can easily imagine situations where it would be advantageous to inline certain calls to a given method but not others, and the current annotation syntax provides no way to indicate this desire. Furthermore, the programmer cannot use the keyword to request inlining on calls to library methods and other code that cannot be modified for external reasons. All this means that even a very clever and careful programmer is limited by the syntax to very coarse-grained control over the inlining decisions that take place. This alone is sufficient motivation to consider adding more expressive inlining annotations to the language, but that is beyond the scope of this paper.

In order to try and combat these difficulties, we've added an "automatic" inlining heuristic that proactively makes some inlining decisions on its own, based on static information cleaned from an analysis of the call graph structure. The inlining policy is a superset of the manual heuristic – that is, it inlines any sites the manual heuristic would inline, which includes calls to methods explicitly marked with the "inline" keyword by the user (and of course the policy is subject to the same call site eligibility restrictions detailed in section 4). Inlining decisions are made on a site-by-site basis and therefore can make more fine-grained choices than possible using only programmer annotations.

The automatic heuristic starts by building a conservative inter-procedural call graph of the program, rooted at the user's main() method[5]. Each node in the call graph uniquely represents a method declaration in the program source, and the directed edges link each method node to the nodes representing methods it could possibly call via one or more call sites (herein lies the conservatism – call sites which cannot be statically bound result in outgoing edges to all possible overrides, even though some may never be possible at execution time). The call graph extends out as far as possible, stopping only at calls to native methods whose bodies are not visible to tc, and specifically is permitted to cross library boundaries into tlib methods, even during a "standard" compilation.

In addition to computing callee edges, we annotate each method node with a number of local attributes – including a rough estimate of the method size (based on a simple accounting of AST nodes), a flag indicating whether the method is selected for code generation under the current compilation mode, and a number of properties related to the graph structure which make subsequent node visits more efficient (such as the list of call sites, back-edges to the caller nodes, and the list of caller sites).

The local node properties allow us to efficiently answer many questions about any method that can be of interest when making inlining decisions – for example, "is this method directly recursive?", or "does this method only have a single caller?". In order to answer more complicated questions, such as "does this call site lie on a mutually recursive cycle?", we solve a simple reverse DFA over the call graph to compute global method reachability. The DFA is setup in the obvious manner (the OUT vector of each

[5] or in the case of a library-only compilation, rooted at every method

node is the union of locally reachable methods and the OUT vector of all callees) and solved using a worklist initialized with a post-order traversal of the graph for efficiency. Each time an inlining operation takes place, the call graph is updated (adding and/or removing edges as necessary) and the solution to the global method reachability DFA is updated incrementally when an edge is removed (the reachability vector is recomputed for the caller node and propagated to its predecessors until a fixed point is re-established).

The inlining strategy is to make a single post-order traversal (as suggested by [6] and others) over the methods in the call graph and visit each of their sites in textual order, applying the inlining policy to make an inlining decision at each site encountered. In the absence of mutually recursive cycles, the post-order traversal constitutes a topological sorting of the call graph, which means that in most cases we shall visit a node only after visiting each of its callees. This is done to maximize the possibility that callees will be leaf methods (contain no call sites) since such methods often represent favorable inlining candidates (inlining a leaf method removes the external call edge from the enclosing basic block and thereby assists intraprocedural analysis). Several idiomatic uses of object-oriented style involve small methods that are either leaves themselves or consist solely of a call to another such small method – this inlining strategy allows entire chains of such methods to be collapsed using a series of inlining operations during a single pass over the call graph (presuming the inlining policy deems it worthwhile).

The automatic inlining policy considers several factors when making the decision of whether or not to inline a given site. Figure 5.2.1 demonstrates the pseudo-code for the policy heuristic as a series of conditions applied sequentially until a decision is reached.

Figure 5.2.1 – Automatic Inlining Policy

```
1. if (site does not meet eligibility constraints) => REJECT
2. if (callee is explicitly or implicitly marked as “inline”) => ACCEPT
3. if (site is the sole caller of this method) => ACCEPT
4. if (callee is a “very small” method) => ACCEPT
5. if (site lies along a mutually-recursive cycle) => REJECT
6. if (callee is a leaf method and caller is library code) => ACCEPT
7. if (callee is a leaf method AND
       operation will not exceed estimated code bloat limit) => ACCEPT
   else => REJECT
```

We now provide some justification for these conditions. Bear in mind that the goal of any inlining heuristic is to: a) minimize the runtime call overhead by removing call instructions from the critical paths, b) maximize the effectiveness of subsequent intraprocedural optimization of the caller body by removing outgoing call edges and, c) minimize the code bloat costs associated with accomplishing goals (a) and (b).

- Condition 1 enforces the constraints discussed in section 4.
- Condition 2 applies the same acceptance criteria discussed in section 5.1, ensuring the sites inlined are a superset of those inlined by the manual heuristic.
- Condition 3 provides that calls to methods which are not called from anywhere else in the program are always inlined – the rationale being that after this inlining

operation, the callee method could effectively be removed from the program, leading to no net change in code bloat as a result of the operation (thus providing benefit (a) with no net cost (c)).
- Condition 4 allows the inlining of methods whose bodies are small enough that the estimated size of the inlined code is comparable to the code size required for the method call itself – this is meant to detect and inline idiomatic cases which often arise in object-oriented programming, such as accessor methods that simply return a constant, get/set a private variable, or act as a simple wrapper around another method call.
- Condition 5 rejects call sites that lie along a mutually recursive cycle. Such call sites do not represent a safety problem for the inliner (condition 1 catches any sites that could potentially lead to non-termination), but inlining them seems unlikely to help us much – by definition inlining such a site would create new sites in the caller (meaning goal (b) cannot be fully realized), and experience suggests most programs written in the imperative style do not employ mutually recursive methods as inner loops (implying little chance of benefit to goal (a)).
- Condition 6 accepts sites that would inline a leaf method into a caller method not currently selected for code generation. Such operations have no direct effect on code bloat (because the resultant caller method will not be code-generated), but serve to collapse chains of library methods into a single leaf method which may be subsequently considered for inlining into user code under condition 7.
- Condition 7 allows for leaf methods to be inlined into user code, provided that doing so would not exceed a global estimated code bloat threshold. Throughout the inlining phase, we track the estimated code bloat that has resulted from inlining operations into methods selected for code generation, and use the information to guide this conditional acceptance. Condition 7 allows for the inlining of reasonable size leaf methods while guarding against code explosion. The current code bloat threshold is an increase of 50%, which seems to be a reasonable limit given the literature.

Section 8 presents the empirical results of applying this criteria, including statistics about which conditions were responsible for the acceptance/rejection of call sites. We’ve done some experimentation to verify that the above heuristic makes reasonable decisions, but by no means have we exhaustively searched the potential design space for the heuristic – there is undoubtedly room for further tweaking and improvement.

6 Structural Inlining Operation

The inlining operation takes place on the abstract syntax tree (AST), the intermediate form used in tc. The AST is a tree-based representation of the Titanium program source, decorated with various attributes such as use-def chains, and symbol-to-declaration edges added by name resolution. Inlining takes place after the AST has been “lowered” to a simple intermediate form analogous to 3-address form where each expression has been broken up into separate statements with temporaries inserted appropriately. The lowered AST form is used by all subsequent phases. The inlining operation was positioned fairly early in the optimization process, notably preceding all the loop optimizations, in order that these subsequent optimizations may benefit from the removal of method call edges by inlining. We now briefly describe the mechanism used to perform the actual inlining operation (the structural inliner) and the issues that arise in its design.

One of the great advantages of placing the structural inlining phase after the lowering phase is that all method call sites we may choose to inline have already been expressed in the basic form shown in Figure 6.1, where each `var` is a local variable, compiler generated temporary, or constant[6]. This includes method calls that began life embedded deeply within some complicated expression – lowering has already resolved all such complexity for us by introducing temporaries as necessary and ordering their computation as required by Java's expression evaluation semantics.

Assuming the call site shown in Figure 6.1 is a call to the method shown in Figure 6.2 (where T1…T4 are types and <exp1>…<exp2> are some expressions), Figure 6.3 demonstrates a source-level representation of the inlined code generated by the structural inliner.

Figure 6.1 – Generic Method Call Site

```
var1 = var2.method(var3, var4, … );
```

Figure 6.2 – Original Method Callee

```
class T2 { …
  public T1 method (T3 fvar3, T4 fvar4) {
      if (…) {
        fvar4 = null;
        return <exp1>;
      }
      else {
        this.x = fvar3;
        return <exp2>;
      }
  }
}
```

Figure 6.3 – Result of Inlining Operation

```
{
  T3 fvar3_u101 = var3;
  T4 fvar4_u102 = var4;
  T1 returnval_u103;
  T2 thistemp_u104 = (T2)var2;
  {
      NULL_CHECK(thistemp_u104);
      if (…) {
        fvar4_u102 = null;
        returnval_u103 = <exp1>;
        goto exit_u105;
      }
      else {
        thistemp_u104.x = fvar3_u101;
        returnval_u103 = <exp2>;
        goto exit_u105;
      }
  }
exit_u105:
  var1 = returnval_u103;
}
```

This small example demonstrates most of the cases that arise in the structural inliner that have to be dealt with properly in order to preserve the semantics of the inlined method call.

The method call is replaced by: 1) a block of code that contains a header with some temporary definitions, 2) a modified copy of the method body and 3) an exit label used for simulating method return. The temporary definitions include a local variable for each formal parameter (which is initialized to the value of the corresponding actual parameter

[6] One or more of the `var`'s may be missing for void and/or static functions

provided in the call site), a local to hold the value of the target object ("this") within the inlined method, and a temporary to receive the return value of the method. The temporary names are augmented with a "uniqueness suffix" which guarantees the name is unique prevents name capture in the generated C code (before introducing this system, we had seemingly endless trouble with name clashes between formals, actuals, and compiler-generated temporaries across modules).

The formal argument temporaries `fvar3_u101` and `fvar4_u102` are necessary in general (despite the fact that lowering guarantees the provided actual parameters will be locals or constants and not expressions) because the callee may choose to modify the value of its formals (as in the assignment `fvar4 = null;`) and such side effects should not be propagated to the caller context. By the same token, the `thistemp_u104` variable is required because the static type of the "this" pointer in the callee context may differ from the static type of the target object provided in the method call (i.e. the static type of `var2` may be a subtype or even a supertype of T2), and the typecast `(T2)` is required for same reason[7]. Although these temps may be required in general, they are seldom necessary in most specific inlining operations – this makes them prime candidates for removal by a subsequent intraprocedural copy-propagation optimization.

A null-check on the value of the target object is inserted at the top of the inlined code when T2 is a reference type to preserve the null-checking semantics of method dispatch in Java.

The inlined form of the method body has been alpha-converted to use the new names that correspond to the formals and "this" pointer in the inlined block. Control flow through the former method body is almost unchanged, with the exception of "return" statements, which have been converted into an assign-and-goto combination (or in the case of void methods, just a goto) which branches to the method exit point to simulate return. For simplicity, we introduced a temporary variable to communicate the returned value from the return points in the inlined body to the exit point where it is assigned to the original call-assign target. This variable could also be removed by a subsequent copy-propagation phase in most cases.

7 Miscellaneous Engineering Issues

The completion of this project was delayed by a long list of engineering issues that arose in the Titanium compiler when inlining was first added – these issues needed to be resolved before the optimization could be considered stable and effective. We document a few of these issues here as illustration.

Because Titanium generates C code, the variable references output may be susceptible to name-capture problems when inlining moves blocks of code into a new context. There were a number of such problems with name clashes between the temporary values

[7] This type-cast is always guaranteed to succeed by virtue of the static binding eligibility constraints, however we don't currently take advantage of this property and may emit a dynamically checked cast in some cases. There may be a small performance advantage to eliding that check.

created during pre-inlining phases such as lowering. These problems were solved by implementing a new code-generation strategy for variable names that ensures uniqueness of names and prevents any possibility of name captures.[8]

One of our goals in this project was to allow inlining of object constructors – they often make ideal candidates for inlining, because they are always statically bound, and are often trivial (especially if they are compiler generated). However, limitations in the previous AST representation of object creation prevented constructor inlining because the operations of object allocation and constructor call were merged in a single AST node. The IR was changed to separate these two steps so that the constructor call could be inlined without affecting space allocation. Inlining of constructors also required restructuring the way we perform instance field initialization – the compilation strategy for these was changed to rewrite the initializations into the beginning of the appropriate constructor to prevent introducing call edges to an external initialization function in C.

The Titanium language spec defines all immutable fields as implicitly final, which means they can only be changed within a constructor. However, the fact that we allow immutable constructor calls to be inlined means that subsequent optimizations may witness immutable fields changing outside the constructor context, which invalidated a number of assumptions made in the intraprocedural analyses. In order to fix this problem (and other unrelated problems which became apparent while working on the fix), it was necessary to re-write a large portion of the def-use analysis to handle immutables correctly and allow for the possibility of constructor inlining.

The inliner allows inlining of calls that cross module and library boundaries. This means that during a "standard" compilation the inliner can and often does decide to pull method bodies out of tlib and inline them into user code, even though those library modules are not currently selected for code generation (they are parsed for type checking purposes, but are not subsequently processed). In order to allow the structural inliner to pull code out of the tlib, the lowering pass was changed from a static compiler phase to an on-demand operation that can be invoked lazily on any module as required by the structural inliner and the call graph creation phase of the automatic inlining heuristic.

The final major problem encountered relates to the optimizations and analysis that follow method inlining. The code bloating effects of method inlining have the potential to create methods much larger and loops much deeper than any programmer is likely to ever write by hand. As a result, the addition of aggressive inlining "stressed out" the subsequent intraprocedural optimizations and analyses in new ways, and revealed several algorithms with nonlinear time or space overhead that broke down in the presence of large method bodies and had to be fixed before the inliner could be used on sizeable applications.

[8] There were several subtle issues in the new uniqueness fix resulting from the fact that inlining may move code across modules – specifically, we had to ensure that such movement wouldn't inadvertently defeat the module caching system used by tcbuild to optimize recompilations of a Titanium application.

8 Results

In order to test the success of our method inliner, we experimented with both inlining heuristics on a number of programs. We present statistics for the benchmarks described in Table 8.1, all of which are "real" scientific applications written by Titanium users.

Name	Description	Command-line Options
gas	2-d gas dynamics hyperbolic solver for the compressible Euler equation	FILE= inputs/paultest
amr	3-d elliptic Poisson solver with variable coefficients using block-structured adaptive mesh refinement	inputs/biginput.1 10
gsrb	2-d Poisson solver with constant coefficients over an infinite domain	-Lx 512 -Ly 512 -I 100
pps	parallel Poisson solver over an infinite domain using finite difference domain decomposition	-L 256 -Np 1 -Nr 4
moose	Microarray oligo-nucleotide selection engine	-X10 -Y5 -s -G100 data/yeast.mgf

Table 8.1 - Benchmark Applications - gas was written by Peter McCorquodale, amr was written by Luigi Semenzato, gsrb and pps were written by Greg Balls, and moose was written by Dan Bonachea. See the Titanium web site [17] for more information about these applications.

All tests were compiled and run on a node of the Berkeley Millennium cluster, with the following configuration:

- 4-way Intel Pentium III Xeon 700 Mhz, 16 KB data cache, 16 KB instruction cache, 1 MB unified L2 cache, 100 MHz system bus speed
- 2 GB physical memory
- Redhat Linux 7.1
- Titanium compiler version 1.790, with tcbuild flags:
  -O --nobcheck --backend sequential --rebuild-tlib
- GNU gcc 3.0 as the C compiler

All tests were done using the sequential backend, since method inlining is a purely sequential optimization and parallel overheads would merely serve to obscure the results.

8.1 Runtime Call Statistics

The method inlining optimization benefits performance directly by removing call overhead and indirectly by enabling subsequent intraprocedural optimization – in both cases we are most concerned with eliminating the frequently-executed call sites from critical paths and inner loops in the program. Consequently, one of the most direct objective metrics for judging the efficacy of an inlining optimization is the count of call instructions removed from the execution trace of the optimized program.

Table 8.1.1 shows the percentage of "inlineable" runtime call statements removed by each heuristic, and the percentage of static call sites in the program text inlined by the heuristic to achieve this result. The rightmost column reports the average runtime call frequency before inlining.

	Manual Heuristic		**Automatic Heuristic**		Avg. inlineable calls/second (before inlining)
	static sites inlined	runtime call reduction	static sites inlined	runtime call reduction	
gas	4.56%	67.07%	18.27%	87.23%	7,264,838
amr	5.14%	4.22%	14.02%	11.14%	4,203
gsrb	3.41%	2.99%	18.13%	3.10%	5,790
pps	2.95%	2.91%	9.59%	3.76%	62,078
moose	7.75%	98.76%	15.01%	99.82%	6,425,519

Table 8.1.1 – Inlining Heuristic Effectiveness

The results are very interesting – the fraction of static sites inlined is relatively steady across the benchmarks (2% to 7% for manual and 9% to 18% for automatic), however the resulting reduction in runtime call statements varies widely.

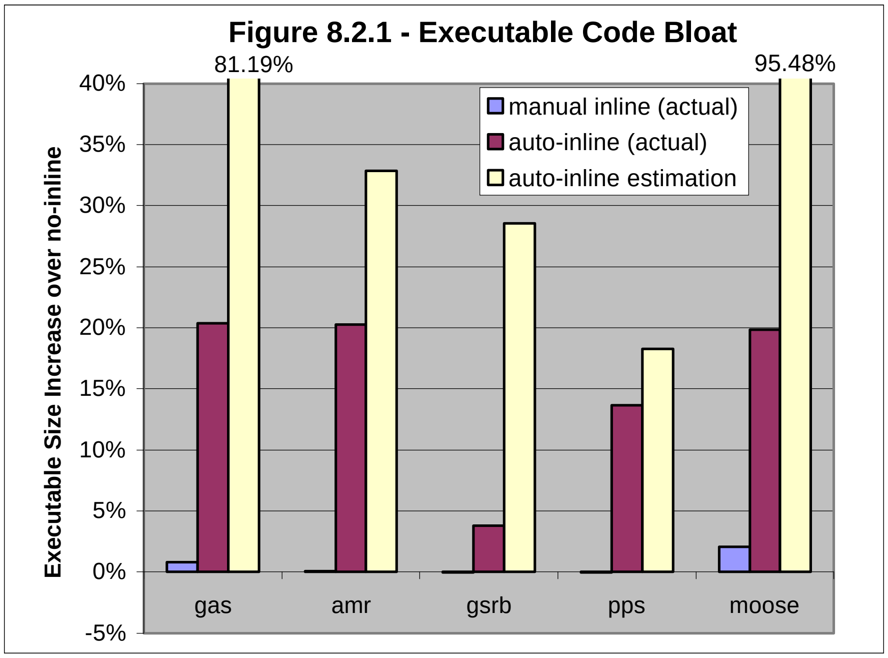
In both gas and moose, the manual inliner has effectively removed a large percentage of the inlineable runtime calls. These two benchmarks have been carefully annotated by the programmer with “inline” keywords (while the others have not), which explains why the manual heuristic has done rather well at removing runtime calls with these two programs. Nevertheless, the automatic heuristic still manages to further reduce the number of runtime calls by a noticeable margin – an indication that we’ve done well in designing the automatic inliner (for comparison, Kaser and Ramakrishnan [14] report the Gnu gcc inlining heuristic removes 0% to 17% of inlineable calls in a suite of C benchmarks with code bloat comparable to ours).

The runtime call reduction for the other three benchmarks is less impressive (although in each case the automatic heuristic does still manage to improve somewhat on the manual heuristic, which in this case is not guided by any explicit “inline” annotations). The explanation for the discrepancy is the design of these benchmarks does not include many inlineable method calls within their inner loops – as the rightmost column of Table 8.1.1 shows, the original runtime call frequency for these benchmarks is 2 to 3 orders of magnitude smaller than for gas and moose. This indicates that method call overhead is not a significant performance factor in these applications and suggests there are probably fewer call site “hot spots” that would make good inlining candidates.

8.2 Code Bloat

The primary cost associated with method inlining is the potential expansion of the final executable due to the duplication of instructions, and the deleterious performance effects that result from this expansion. The automatic inlining heuristic uses a running estimate of the code bloat to guide inlining decisions.

Figure 8.2.1 shows the actual code bloat resulting from compiling our benchmarks with each of the inlining heuristics, and the code bloat that was estimated by the heuristic during compilation. All quantities are shown as a percent growth over the version without inlining.



The code bloat from the manual inline heuristic was uniformly low – all below 3%. In gsrb and pps, the executable was actually slightly smaller under the manual inlining heuristic, which is probably due to the inlining of trivial methods that optimize away to nothing and therefore consume less space than the instructions comprising the former call site.

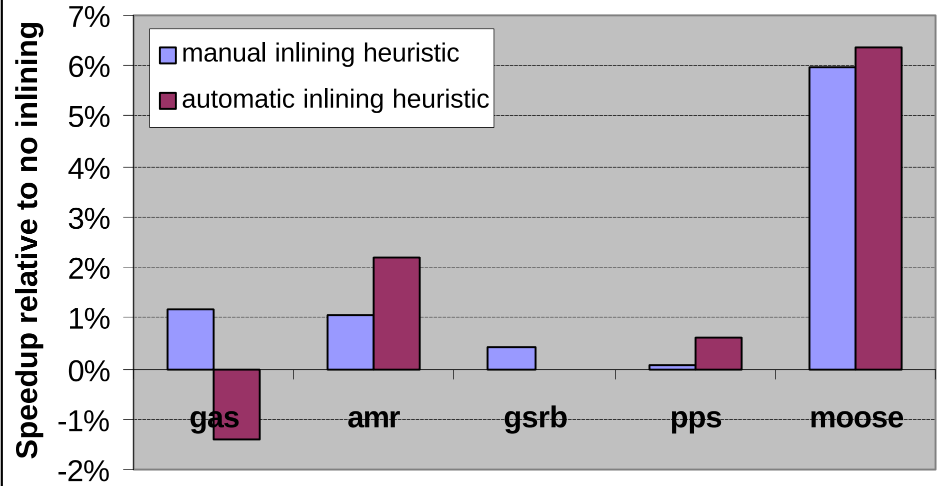
The code bloat for the automatic heuristic is more noticeable (all under about 20%), but still well within reasonable limits– the literature reports some production inliners allow up to 200% code expansion. Of course, the code bloat of both heuristics could be expected to be higher with more liberal use of the "inline" method declaration modifier. We should mention that although the code bloat observed under the automatic heuristic was modest, the increase in compile time for post-inlining optimizations was very noticeable in a few cases – for example, "gas" took over an hour to optimize a few of the larger method bodies generated by the automatic inliner. Clearly the post-inlining intraprocedural optimizations could use further tuning to efficiently handle the large methods created by the inliner.

We observed the code bloat estimations computed by the automatic heuristic to be overly high in most cases, and feel there are three primary explanations for this. First, the current size estimation technique is very crude – it amounts to counting the number of AST nodes in a given method body. However, there is only a weak correlation between AST node code and number of lines of C code generated. A smarter technique that weighed different categories of AST nodes differently based on a better approximation of how much code they produce might give better results. Secondly, the estimate of the code bloat is based only on the contents of live Titanium method bodies, while the real executable also contains code for dead methods in all live classes and the object code for the Titanium runtime system – this overhead reduces the proportional impact of the inlining operations and makes the size estimate appear inflated. Finally, the relation between source text size growth and object code size growth is usually non-linear due to increased optimization opportunities created by inlining – Cooper et al. [6] confirmed this observation over a number of compilers in their source-to-source inliner.

8.3 Execution Time Speedup

Figure 8.3.1 reports the effects of the two inlining heuristics on the execution time performance of the benchmarks.

Figure 8.3.1 - Total Execution Time Speedup



In interpreting these results, it is instructive to have a basis for comparison on results achieved by other inliners. Cooper, Hall and Torczon [6] report a performance improvement of 12% to 28% attributed to inlining on a number of systems surveyed. Other similar inlining systems report results of comparable magnitude. Leupers and Marwedel [16] use a profiling-based exhaustive search to discover an optimal set of inlining decisions, and report a performance speedup that peaks at 33%. It seems that inlining rarely provides earth-shattering speedups, but speedups in the neighborhood of 15% to 25% are often achievable.

Unfortunately, the realized speedups for our inliner are somewhat less than one might hope. Keeping in mind that amr, gsrb and pps are not good candidates for inlining (as discussed in section 8.1 above) and can’t be expected to show a very noticeable change, we turn our attention to gas and moose. moose shows a moderate but respectable speedup of 6% to 7% under both heuristics. gas doesn’t gain very much from the manual heuristic, and in fact suffers a slow-down when the automatic heuristic is used. Closer inspection suggests that the problem lies in the fact that the “workhorse” methods in gas are quite large to start with, and they become truly enormous once the automatic heuristic is applied. We believe the subsequent optimization phases (especially in the C compiler) have trouble effectively optimizing such large methods with so many local variables and the result is poorly optimized object code. It is also likely that the large size of these critical methods lead to increased I-cache misses at runtime that further exacerbate the problem.

Cooper, Hall and Torczon [6] report that in a number of optimizing compilers the performance effects of inlining can be often be limited or even detrimental due to increased register pressure in the caller and the common inability for subsequent data-flow analyses to deal with an increase in the number of local temporary variables over a certain hard-coded limit. We feel that these effects, combined with the lack of several important post-inlining sequential optimizations in the Titanium optimizer are responsible for masking the performance improvements offered by our method inliner.

Further work on the subsequent optimization phases seems to be required in order to fully realize the speedups made possible by our method inliner.

8.4 Automatic Heuristic Policy Decision Breakdown

The inlining policy in the automatic heuristic uses a number of criteria to proactively decide whether a given call site should be inlined. Tables 8.4.1 and 8.4.2 show a statistical breakdown of why the 4463 call sites considered while processing the moose benchmark were accepted or rejected, respectively. See section 5.2 for more explanation of these categories.

Reason site was accepted	Number of sites	Fraction of total sites
explicitly marked using inline modifier by the user	93	2.08%
implicitly marked as inline	34	0.76%
single caller	225	5.04%
very small callee	227	5.09%
leaf method called from library code	0	0%
leaf method within code bloat limit	12	0.27%

Table 8.4.1 – Accepted Call Sites Inlined in moose

Reason site was rejected	Number of sites	Fraction of total sites
uncertain callee	38	0.85%
native method	884	19.81%
directly recursive callee	188	4.21%
mutually recursive site	0	0%
code bloat limit exceeded	25	0.56%
default rejection	2737	61.33%

Table 8.4.2 – Rejected Call Sites in moose

The first two acceptance categories correspond to the manual heuristic, and those 127 sites are responsible for the 98.7% reduction in runtime calls reported in section 8.1. The remaining acceptance categories are responsible for the additional improvement realized by the automatic heuristic. The first three rejection categories indicate sites which are never eligible for inlining. It is interesting how many potential inlining sites are lost to native method calls – this seems sufficient justification to consider alternate means for allowing the inlining of native code. Even though the final code bloat estimate for moose (93% growth) significantly exceeds the bloat threshold of 50% growth, this restriction was only responsible for the rejection of 25 sites calling leaf methods (sites in the first five acceptance categories are not hindered by the code bloat restriction).

9 Related Work on Method Inlining

There is a fairly large body of related work on inlining, as it is one of the most successful and tractable forms of interprocedural optimization. Inlining has become especially important in recent years with the emergence of call-intensive object-oriented languages as a popular development platform for production systems. We discuss some of the more interesting research here that hasn't already been mentioned elsewhere in this paper.

Kaser and Ramakrishnan [14] investigate the relative effectiveness of inlining policies that inline different method versions. (For example, when inlining a method call, there is an implicit choice whether to inline the original version of the callee, or the "current" version of the callee, which may have been changed by previous inlining operations. All previous research in method inlining has implicitly focused on current-version inlining.) They prove that original-version inlining is theoretically more flexible and powerful than current-version inlining by any reasonable metric, however in practice current-version inlining tends to perform better because the inlining strategy heuristic requires more look-ahead in order to perform adequately under an original-version inlining policy. Our inliner uses current-version inlining.

Dean and Chambers [8] present an interesting technique called inlining trials, where the decision space is searched by tentatively applying each inlining operation and subsequently optimizing the result. The optimizer then estimates the size and running time of the resulting optimized code and provides this input to a cost/benefit analysis that decides whether to accept or reject the proposed inlining operation.

Ayers, Gottlieb and Schooler [2] present an inlining strategy that assigns benefits and costs to all the inlining candidate sites (based on estimated call frequency and estimated increase in compile time for subsequent optimization), then sorts the candidates by decreasing benefit and considers each in turn. Such a strategy seems worthwhile because it ranks sites globally and starts with those perceived to be most crucial, as opposed to the traditional "post-order traversal over the call graph" approach that often arbitrarily orders unrelated sites.

Several research projects (Ashley [1], Jagannathan and Wright [8]) have shown the value of flow-directed static analyses for making inlining decisions in functional languages such as Scheme, where function calls are pervasive and inlining is an absolutely essential optimization. It is unclear whether the direct application of these techniques would be as effective in an imperative programming language such as Java, but the possibilities seem hopeful.

Cooper, Hall and Kennedy [4],[5] discuss a related optimization technique called "procedure cloning", whereby the compiler generates several versions of a given procedure, optimized using different assumptions about the input parameters. Then, the various calls to the procedure are partitioned amongst the corresponding versions based on the properties of the actual parameters. This technique provides many of the indirect benefits of inlining (better subsequent optimizations) with substantially reduced code bloat. This technique was not considered for Titanium because using it effectively requires fairly sophisticated interprocedural analyses to make good cloning decisions and interprocedural optimizations to take advantage of them that are not present in the current optimizer. Conversely, inlining provides benefits to purely intraprocedural optimizations, which are present in our current optimizer.

10 Future work

10.1 Heuristic Improvements

A number of studies on method inlining have demonstrated that feedback-directed inlining heuristics can be very effective at reducing the dynamic call frequencies of most real programs, leading to corresponding speedups. Such techniques based on profiling information almost uniformly outperform the best available static inlining heuristics. Titanium doesn’t currently have a mechanism for feedback-directed optimization, but method inlining is a compelling reason to develop one.

Some research has also been done on estimating the dynamic call frequency of program call sites in imperative languages using static analyses based on structural aspects of the program (for example, program loop structure and branches). Wagner, Maverick, Graham and Harrison [18] present a static analysis system capable of estimating the busiest ¼ of call sites with 76% accuracy in C programs – the output of such an analysis could clearly be used to help direct the inlining strategy and improve the quality of inlining decisions.

10.2 Inlining of general native methods

We added a special-case hack to the code generation phase that allows the small native method called by the java.lang.Object constructor to be inlined, but there are other small, heavily used native methods in the language that could potentially benefit from inlining (for example, the native methods which implement the trigonometric functions in java.lang.Math by simply calling the C library equivalent). Because of separate compilation, it is unlikely that any C optimizer could successfully inline these external calls to the native methods – it is especially unfortunate in the case of the trigonometric functions, because the calls to these C library functions are often recognized and specially handled by C optimizers (including gcc). It would be nice if the Titanium compiler could somehow be modified to allow inlining of native methods – teaching tc to actually read and parse the native code would be a very difficult task, but it may be possible to modify our code generation process so the native method bodies are available to the C optimizer/inliner whenever the calls to these methods are encountered.

10.3 Dead-method Elimination

One important piece of information that can easily be gleaned from the interprocedural call graph that is built by the more advanced inlining heuristic is a conservative set of all the live methods in the program. Given this information, a reasonable optimization to consider is dead method elimination, whereby the method bodies which are found to be unreachable are omitted at code generation time – thereby improving compile time and reducing the size of the final executable. This optimization would also naturally complement method inlining, because methods whose direct call sites have all been inlined become dead and can be removed (this also means that methods with only a single caller site can always be inlined with no possibility of code bloat). As a result, we did a thorough investigation into the viability of implementing this optimization.

This optimization is especially attractive when performing a rebuild-tlib compilation, because only the library methods that are actually referenced will be included in the final executable. This optimization is less useful during a standard compilation, because most UNIX linkers will be unable to omit unused library objects (due to the structure of the C code generated for the tlib) and will end up linking the application with a complete copy of the tlib library (this unfortunately means that even tiny Titanium programs can result in sizeable on-disk executables – our compilation model is clearly targeted for creating large, long-running scientific applications).

Unfortunately, several factors conspire to make dead-method elimination impractical within our current system architecture. The primary difficulty is that method up-calls from hand-written native C code to generated methods are not visible to tc, and therefore those methods might erroneously be eliminated by the optimization even though they are still live (leading to application link errors or possibly just runtime crashes). While it may be possible to hard-code a list of permanently live tlib library methods into tc (similar to what was done to solve a similar problem in the type qualification inference system), this would not solve the problem for calls from user-provided native methods, which are included in some of our important applications.

The second problem with implementing dead-method elimination is a slight mismatch between the interprocedural call graph currently generated by the automatic inlining heuristic (which is rooted at the program's main method and transitively branches out to any reachable methods), and the call graph required to safely compute method liveness. In order to avoid inadvertently removing a live method, the call graph must also include all the static initializers of live classes in the root set used to compute the call graph. This is not currently done for inlining because computing class liveness is non-trivial and in most real Titanium applications the static initializers consume a negligible fraction of the total execution time, making optimization of this code unprofitable.

Finally, the fact that dead-method elimination would be ineffective under the "standard" compilation model (without --rebuild-tlib) makes it not worthwhile to implement at this time. However, we feel our time was well spent investigating and documenting these issues, because it is conceivable that future changes to the compiler architecture might make this optimization practical at some time.

10.4 Other Miscellaneous Improvements

The current inliner allows immutable creation to be fully inlined (with no calls to external functions in the generated C code). Non-immutable object creation can be almost fully inlined, with the exception of a call to the storage allocator (which is usually an entry point in the Boehm-Weiser garbage collector). The Boehm-Weiser GC offers macros for fully inlining the common-case allocation (where space is available and no collection is necessary) – this could potentially further cut down the function-call overhead involved in common-case object creation, but is unlikely to help much with the intraprocedural optimization of surrounding code since the possibility of a call to an external function still exists. Nevertheless, further exploration may be warranted for object allocations within inner loops.

Cooper, Hall and Torczon [7] report an interesting case where performance of a numerically intensive Fortran benchmark was adversely affected by an increase in the number of floating-point stalls. The explanation turned out to be related to Fortran's parameter aliasing rules, which forbid any overlap between arrays passed as parameters to a function. When a few critical calls in the numerical kernel were inlined, the aliasing assertions that previously existed at the call boundary were lost, leading to poorer subsequent analysis and less effective instruction scheduling for the pipelined floating point unit. This scenario is relevant to Titanium because it has analogous rules forbidding overlap of Titanium array parameters in the absence of the "overlap" method annotation, which makes it legal. We haven't knowingly encountered the scenario described by Cooper et al., however it seems advisable to prevent such problems in the future by inserting the appropriate "nooverlap" annotations at inlined call sites involving array parameters so that subsequent intraprocedural analyses can still leverage the aliasing information provided by the former call boundary.

The Titanium optimizer is lacking several standard intraprocedural sequential optimizations, most notably copy-propagation and common-subexpression elimination – these optimizations are very important for maximizing the benefit of an inlining operation. Inlining a method call creates a number of temporary variables, some of which may be unnecessary in specific cases, and often exposes significant optimization opportunities. Unfortunately, the subsequent phases don't fully take advantage of these opportunities at present. These optimizations are especially important to apply at the Titanium level, because experience has shown that C optimizers tend to poorly optimize the code we generate for immutable types and global pointers, which frequently involves aggressive use of structs at the C level. We feel that the implementation of a few such optimizations would greatly improve the performance benefit of inlining.

11 Bibliography

1. Ashley, Michael J. “The Effectiveness of Flow Analysis for Inlining”, ICFP (1997).
2. Ayers, Andrew, Gottlieb, Robert and Schooler, Richard. “Aggressive Inlining”, PLDI (1997).
3. Chiba, Yubi. “Devirtualization Techniques in a Static Compiler for Java”, Systems Development Laboratory, Hitachi Co Ltd.
4. Cooper, Keith D., Hall, Mary W., and Kennedy, Ken. “Procedure Cloning”, IEEE (1992).
5. Cooper, Keith D., Hall, Mary W., and Kennedy, Ken. “A Methodology For Procedure Cloning”, Computer Languages (1993).
6. Cooper, Keith D., Hall, Mary W. and Torczon, Linda. “An Experiment with Inline Substitution”, Software – Practice and Experience, (June 1991).
7. Cooper, Keith D., Hall, Mary W. and Torczon, Linda. “Unexpected Side Effects of Inline Substitution: A Case Study”, ACM Letters on Programming Languages and Systems (March 1992).
8. Dean, Jeffrey and Chambers, Craig. “Towards Better Inlining Decisions Using Inlining Trials”, ACM LISP (1994).
9. Dean, Jeffrey, Chambers, Craig and Grove, David. “Selective Specialization for Object-Oriented Languages”, ACM SIGPLAN 95.
10. Ghemawat, Sanjay, Randall, Keith H., and Scales, Daniel J. “Field Analysis: Getting Useful and Low-cost Interprocedural Information”, ACM PLDI (2000).
11. Hilfinger, Paul. “Titanium Language Working Sketch”. http://www.cs.berkeley.edu/Research/Projects/titanium/doc/lang-ref.ps
12. Ishazaki, Kazuaki et al. “Design, Implementation and Evaluation of Optimizations in a Just-In-Time Compiler”, IBM Tokyo Research Laboratory.
13. Jagannathan, Suresh and Wright, Andrew. “Flow-directed Inlining”, PLDI (1996).
14. Kaser, Owen and Ramakrishnan C.R.. “Evaluating Inlining Techniques”, Computer Languages 24 (1998).
15. Kaser, Owen, Ramakrishnan, C.R. and Pawagi, Shaunak. “On the Conversion of Indirect to Direct Recursion”, ACM Letters on Programming Languages and Systems (1993).
16. Leupers, Rainer and Marwedel, Peter. “Function Inlining under Code Size Constraints for Embedded Processors”, IEEE (1999).
17. Titanium web site. http://www.cs.berkeley.edu/Research/Projects/titanium
18. Wagner, Tim A., Maverick, Vance, Graham, Susan L. and Harrison, Michael A.”Accurate Static Estimators for Program Optimization”, ACM SIGPLAN (1994).
19. Yu, Ting and Kaser, Owen. “A Note on ‘On the Conversion of Indirect to Direct Recursion’”, ACM Letters on Programming Languages and Systems (1997).