An Evaluation of Automatic Object Inline Allocation Techniques

Julian Dolby
Department of Computer Science
University of Illinois at Urbana
dolby@cs.uiuc.edu

Andrew A. Chien
Department of Computer Science and Engineering
University of California, San Diego
achien@cs.ucsd.edu

Abstract

Object-oriented languages such as Java and Smalltalk provide a uniform object reference model, allowing objects to be conveniently shared. If implemented directly, these uniform reference models can suffer in efficiency due to additional memory dereferences and memory management operations. Automatic *inline allocation* of child objects within parent objects can reduce overheads of heap-allocated pointer-referenced objects.

We present three compiler analyses to identify inlinable fields by tracking accesses to heap objects. These analyses span a range from local data flow to adaptive whole-program, flow-sensitive inter-procedural analysis. We measure their cost and effectiveness on a suite of moderate-sized C++ programs (up to 30,000 lines including libraries). We show that aggressive inter-procedural analysis is required to enable object inlining, and our adaptive inter-procedural analysis [23] computes precise information efficiently. Object inlining eliminates typically 40% of object accesses and allocations (improving performance up to 50%). Furthermore,

1 Introduction

Object-oriented languages provide *abstraction*, allowing programmers to isolate conceptual portions of a given program behind opaque interfaces, with attendant benefits in code modularity and reusability. Languages such as Java [32], Lisp [30], Pool [2] and Sather [31] provide this abstraction with opaque objects for which clients have a reference and an interface specification. This isolates the clients from any changes in a given object's implementation. Even fine-grained portions of a program, such as individual points for a graphics library, can be conveniently expressed in this manner.

But these same interfaces create overhead if implemented in the manner of a traditional Lisp or Java runtime system, using dynamic dispatch to call methods and heap-allocated objects accessed via pointers. Additionally, an object-oriented programming style generally encourages the use of small methods and objects [5]. The combination of small methods and dynamic dispatch is a well-studied problem: dynamic dispatches are optimized statically by type inference [1, 6, 21, 24], dynamically by inline caching [16] or with hybrid approaches like type feedback [17]. Static or hybrid type analysis has been combined with method specialization [8, 25] to allow inlining, removing the small functions common in object-oriented code.

Pervasive use of heap-allocated objects introduces overhead for memory management and repeated pointer dereference. This can both increase memory traffic and hurt local code efficiency by reducing opportunities for register allocation which inhibits many scalar optimizations. Pointer dereference (called *pointer chasing*) overhead not only incurs additional memory traffic, but given performance sensitivity to data locality, typically reduces cache efficiency. This topic has been studied by many researchers, using both runtime techniques (e.g. fine grained multi threading [22]) and compile time approaches (e.g. representations that explicate dependencies thru pointers [19]). But it remains a challenging open problem. *Object inlining* coalesces objects by inline allocating child objects within their container objects. This attacks pointer chasing by eliding the pointers and converts an unpredictable memory reference into one with spatial locality.

But object inlining poses challenges of its own: it requires an analysis capable of distinguishing individual container and containee objects, both to ensure that merging them does not change sharing relationships, and to generate appropriate code for accessing state of merged objects where that is needed. A whole-program analysis that does this was presented in [9], and showed speedups of up to three-fold on a set of object-intensive benchmarks. However, that study did not assess much analysis power is really required, and how many fields can be inlined on a wider range of benchmarks. In this paper, we address those questions.

So to assess the feasibility and benefit of object inlining, we study its effectiveness using several analysis frameworks of varying power and cost, and a benchmark suite including the NIHCL [14] and OATH class libraries, which together provide multiple implementations of a range of common data structures. These codes range from a few hundred lines to over 10,000 lines of C++ code (plus 20,000 lines of library). We implemented three different program analyses. They all use data-flow properties to track how object fields are used

OOPSLA '98 10/98 Vancouver, B.C.

and defined; the analysis frameworks employed are local data flow, traditional control flow analysis [27] and adaptive flow analysis [24]. The control flow analysis and adaptive analysis variants are based upon the techniques in our prior work [9]; however, our study revealed deficiencies in those techniques so we generalized them substantially for this study.

Our results indicate object inlining optimizations eliminate typically 40% and as much as 90% of the object accesses and allocations, and can deliver significant performance benefits (averaging 10% faster but ranging from no improvement to 50%). However, reaping these benefits requires powerful inter-procedural analysis that must focus effort to avoid excessive cost. Both the simple local technique nor the traditional flow analysis proved insufficent. Fortunately, the adaptive interprocedural analysis we employed [24] computes precise information efficiently.

We begin by discussing our approach to inline allocation in Section 2 and presenting an example program in Section 3. Next, Section 4 describes the Concert System [7] in which work was done, and Section 5 details our three program analyses. These analyses are evaluated on a suite of C++ programs in Section 6. This suite is summarized in Section 6.2 and performance metrics and results are given in Sections 6.3 and 6.4. Finally, we contrast related work in Section 7 and conclude with Section 8.

2 Automatic Object Inlining

The idea behind automatic object-inlining is provide a more efficient implementation without altering the model seen by the programmer. Thus, the source program might describe a logical structure (which we call the *literal structure*) of several objects, connected by references, and the resulting implementation after object inlining might be these objects fused into a single object. The literal and optimized data structure implementations for an example pair of classes is illustrated by Figure 1.

The literal implementation requires multiple memory operations to allocate and reclaim the storage for the objects, and pointer dereferences to reach the `Point` objects. In contrast, the optimized implementation can be allocated and reclaimed in a single operation and the fields directly accessed with a single load with offset instruction.

2.1 Analysis and Transformation Requirements

For inlining to be semantics preserving[1], program analysis must provide two pieces of information for each inlinable field. First, the analysis must precisely identify all accesses to the child object, and, second, to ensure that sharing relationships are correctly preserved, the analysis must ensure that the child is not stored into multiple parents via the given field. The need for precision is significant and difficult to achieve in many cases. Thus, the goal of the analysis is to identify the object fields (and contours[2]) for which the object inlining transformation is semantics preserving. More successful analyses will find more field, contour pairs which can be inlined.

If the analysis produces the requisite information, then the object inlining transformation consists of the following steps[3]:

1. Create a new definition for the parent object which includes the inlined child object (fields and methods), including constructor/destructor methods
2. Modify the allocation points to use the new object definitions
3. Rewrite all accesses to the child object's state as accesses to the inlined child object.

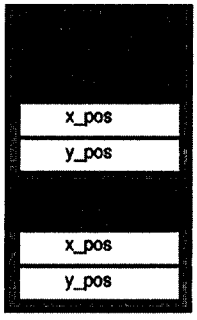
2.2 Explicit Inline Allocation

Some programming languages [10, 33], notably C++, allow programmers to manually specify inline object allocation to improve performance. As with the automatic approaches, the objective is to reduce storage management overhead as well as the number of pointer dereferences required to execute a program. An example of explicit inline allocation is shown in Figure 2.

```
class Rectangle {
  // Points are inlined
  Point upper_left;
  Point lower_right; };

class Point {
  int x_pos;
  int y_pos; };
```

C++ Source



C++ Implementation

Figure 2: Explicit Inline Allocation in C++

In our explicitly inlined example, there are still multiple objects. References to child objects allocated inside parents are allowed.

There are two basic differences between automatic and explicit object inlining: whether inline allocation is visible to the programmer—that is, whether it is part of the programming model—and whether the parent and child objects are fused. As can be seen in Figure 2, explicit inline allocation requires the programmer to explicitly indicate which of the child objects are to be inlined. This requires explicit effort, and a change to the code structure by the programmer. Further, it is not a semantics-preserving transformation in general – as the the inlining operation changes the sharing semantics. The inlined child objects are by-value whereas outlined objects are by reference. Thus the advantages

[1] The transformation must be semantics preserving in order for the program to continue to correctly implement the source program.

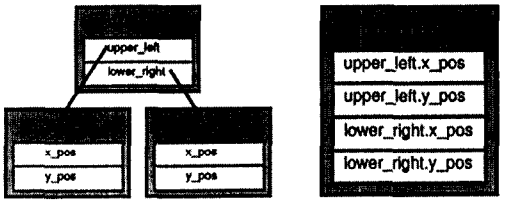
[2] The separately customizable contexts of use of the objects.

[3] Once a given field has been found inlinable, a policy decision must determine whether or not it actually is inlined. Since we are exploring the feasibility of object inlining, we currently inline whenever possible.

```
class Rectangle {
  // '*' declares references
  Point *upper_left;
  Point *lower_right; };

class Point {
  int x_pos;
  int y_pos; };
```

C++ Source



Literal Inlined (Optimized)

Figure 1: Literal and Optimized Implementations

of automatic object inlining is that it can provide the performance benefits without requiring programmer effort and that automation ensures the correctness of the inlining transformation (and "undoes it" should the addition of code invalidate the transformation).

Fusing parent and child objects into a single layout has two advantages. First, it allows optimizing per-object operations, such as concurrency control in a concurrent object-oriented model (this was one of our original motivations). Second, it enables code generation for targets, such as the Java Virtual Machine and the Concert runtime system, that do not permit interior pointers.[4]

These challenges will be illustrated by our running example, introduced in Section 3.

3 An Example

To provide continuity, we employ a single example throughout for exposition of our analyses. The code example consists of the two class definitions from Figure 1, and some methods (Figure 3). A `Rectangle` is defined by two `Points`, each of which in turn consists of two integer coordinate values in 2-dimensional Cartesian space.

The example methods and `main()` program create several `Points` and a `Rectangle`, checking the validity of the rectangle (lower right corner is really right of and below the other corner). Finally, the program prints the x coordinates of two points.

The example illustrates the analysis requirements. To safely inline the `Points` (`p`, `q`) in the `Rectangle` (`r`), all uses of p and q must be identified. This includes all of the uses within the `Rectangle` class's methods and the main program. In addition, the analysis must determine the sharing properties of the `Points` relative to its use in the Rectangle. This is required because inlined objects have by-value semantics, that is, they cannot be shared by multiple parents thru inlined fields.

Note that the `Point` class cannot be inlined indiscriminately, and thus the analysis must identify the sets of creations of `Points` which are to be optimized. Creations of p and q must be deleted, and their constructors redirected to the corresponding inlined fields of r. The two constructors for the inlined versions of p and q must be specialized differently as one works upon the `lower_left` and on the `upper_right` point. Finally, the creation of s must be left alone.

Automatic inlining analysis must determine the sharing properties of the object to be inlined to determine unambiguously which child object is assigned into which container. Since p, q and r are created in the same block of code, the relationship is apparent: the instances p and q are assigned into r. However, in general these creations could be separated by function calls and even assignments thru global state.

Accesses to the objects must also be transformed. For example, the `cout <<` statement in `main` gets two `x_pos` values, one from a `Point` inlined into `upper_left` and one from a free-standing `Point`. These two `x_pos` operations must have different implementations, as one must access the inlined field and the other a free-standing one.

```
Point::belowRight?(p) {
  if (p.x_pos > this->x_pos)
   return (p.y_pos < this->y_pos);
  else
   return false;
}

Point::Point(x, y) {
  this->x_pos = x;
  this->y_pos = y;
}

Rectangle::Rectangle(ul, lr) {
  if (lr.belowRight?(ul)) {
    this->upper_left = ul;
    this->lower_right = lr;
  } else
    error("invalid rectangle");
}

main {
  p = new Point(3, 8);
  q = new Point(8, 6);
  r = new Rectangle(p, q);
  s = new Point(8, 7);
  cout << r.upper_left.x_pos << s.x_pos;
}
```

Figure 3: Example Methods

[4] Additionally, some issues such as garbage collection are more difficult – but not impossible – with interior pointers.

4 Background: The Concert Compiler

The implementation of our object inlining analyses was done in the Illinois Concert System [7], and so a brief discussion of the relevant aspects of the system is given here to provide context for subsequent description of the optimization. Most relevant is the program representation and the analysis and cloning frameworks, all discussed below.

4.1 Program Representation

The primary program representation used by the Concert Compiler is the Program Dependence Graph (PDG [12]) in Static Single Assignment (SSA) form, of which a brief sketch is provided here mostly to introduce terminology we use while describing our analyses. Figure 4 shows an example PDG fragment from the `Point::belowRight?` method from Figure 3.

The PDG represents methods as a tree of control dependence regions and conditional (including loop) nodes. Each region is the child of the conditional node that governs whether or not it executes, so loop nodes have one child region for the body and if nodes have one child each for true and false branches. The phi nodes of SSA form are attached to these conditional nodes. every other node – our graph has function calls, primitive operations and field accesses – is contained in the region of the conditional governing its execution. Within a single region, ordering between nodes is represented explicitly by a set of data constraints.

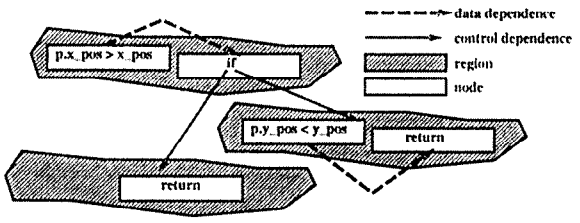


Figure 4: Program Dependence Graph

Note how dominance and post-dominance is implicit in this representation: if `p.y_pos < y_pos` executes then `p.x_pos > x_pos` must already have done so because it is in a parent region and has a data dependence. We use this property for our inlining analyses. In our subsequent discussion, we use aspects of this representation which we formalize below. For single nodes, we use their *region* and *kind* properties and, for pairs of nodes within a single region, we use the ordering constraint $\succ$:

$$region(n) \leftarrow \text{the region containing } n$$

$$n_1 \succ n_2 \leftarrow n_1 \text{ must execute after } n_2 \text{ does}$$

$$kind(n) \leftarrow \begin{cases} if & \text{for conditional nodes} \\ while & \text{for loop nodes} \\ phi & \text{for all phi nodes} \\ call & \text{for function calls} \\ access & \text{for all field accesses} \\ creation & \text{for } \texttt{new} \text{ statements} \\ primitive & \text{for all other primitives} \end{cases}$$

For regions, we query their parent conditional, i.e. the conditional that controls whether or not the nodes within the region execute:

$$parent(r) \leftarrow \text{the conditional node governing } r$$

For individual SSA values, we require knowing their *creation*, which is the node responsible for generating that value, and their *reaching definitions*. We assume that unnecessary moves are eliminated, so the kind of *creation*(v) be one of *call, phi, access, primitive* or *creation*.

$$creation(n) \leftarrow \text{the node which creates } n$$

$$reaching(n) \leftarrow \text{the set of nodes using } n$$

4.2 The Analysis Framework

The Concert compiler has a global analysis framework – adaptive analysis [24, 23] – that performs context sensitive flow analysis. The flow analysis ultimately builds a program-wide data-flow graph connecting the values within and across the individual method program graphs, with those values being specialized as needed by context sensitivity. Context sensitivity adapts to program structure, focusing analysis effort on interesting portions of the program.

The unit of context sensitivity is the *contour* [29], each of which represents an execution environment. For a given method, *method contours* can discriminate arbitrary data-flow properties of its *caller* and *creator*:

caller – the calling statement and contour. This covers arguments, allowing discrimination based upon data-flow properties of caller and its arguments.

creator – the object contour representing `self`. This permits a limited form of alias analysis based upon properties of the target object.

An *object contour* represents a set of method contours of statements that create a given object. That is, each `new` statement is analyzed with some number of method contours, and the object contours corresponding to that new statement each group some set of those method contours. Thus, an object contour represents a `new` statement called in some context.

In traditional control flow analysis (nCFA), contours are statically created to analyze a method separately for different callers from one or more level. But in adaptive analysis, contours are created and split on demand: they are created when the analysis needs to distinguish some property. An initial coarse data flow graph is built and then scanned for imprecisions; these imprecisions are used to direct selective adding of contours – splitting existing ones – to improve information quality. This process iterates until no more contours improve information.

The original use of this framework was type inference, which creates contours to distinguish type information. Method contours are created for different sets of argument types; for polymorphic fields, different object contours are built for the containing object to differentiate the types in the field. The analysis framework

includes a mechanism for distinguishing object contours with respect to uses of objects. We also use this framework to implement the object-inlining analysis.

Figure 5 illustrates analysis on the program fragment from Figures 1 and 3. For simplicity, we ignore the last two statements in `main`. Figure 5(a) illustrates the initial coarse graph. In the example, there is one contour per method and per class: the contours in the figures are labeled with the function and *(m,o)*, which records the method contour and object contour numbers. In Figure 5(a), the object contour 0 represents `Points` and 1 represents `Rectangles`. The `main` function has - as its object contour because it is not a method on any object.

Object inlining analysis needs to distinguish objects assigned into different fields, and currently objects from contour 0 are assigned into but `upper_left` and `lower_right`. So the demand driven specialization mechanism tracks the values assigned to these slots back to the creations of `p` and `q`; it then splits contour 0 into two contours – 0 and 2 – to distinguish these two creations. This also causes all method contours to be split so that `this` always has one contour. The resultant refined graph is shown in Figure 5(b).

Subsequent discussion of our analysis distinguishes specialized values by subscripting them with a given contour, so that v_c is the value v specialized to contour c. Furthermore, we use the following aspects of our representation:

$$
\begin{array}{rcl}
v_{1c_1} \succ v_{2c_2} & : & \text{data may flow from } v_{1c_1} \text{ to } v_{2c_2} \\
v_{1c_1} \prec v_{2c_2} & : & \text{data may flow from } v_{2c_2} \text{ to } v_{1c_1} \\
Creators(v_c) & \leftarrow & \text{object contours of } v_c. \\
self(c) & \leftarrow & \text{object contour of } \texttt{this}_c
\end{array}
$$

Recall the object contours represent specializations of classes, so $Creators(v_c)$ is essentially the type of v_c.

5 Analysis

We explore three different analyses for automatic object inlining, each of increasing analysis power. Recall that the goal of object inlining analysis is to identify the object fields (and contours (See Section 4.2)) for which the object inlining transformation is semantics preserving. More successful analyses will find more field, contour pairs which can be inlined. For a field to be safely inlinable, the analysis must be able to precisely enumerate uses of the child object, and the sharing relationship between the parent and child objects. These properties are are formalized in Section 5.1. The three different analyses for inlining, each of increasing power, are described in subsequent sections.

5.1 Criteria for Inlinable Fields

To prove a field is safely inlinable, an object inlining analysis must compute the following information. All of the results must be precise. Let f denote a single field in a contour for a single class.

R_f: the set of references to a container object of field f

E_f: the set of references to a containee object of field f

$Up_f : E_f \longmapsto R_f$: a map from containees to their corresponding containers

Computing these properties requires precise resolution of control flow and data flow in large programs. These definitions are independent of the program representation used, so the exact meaning of *reference* and *class* depend the analysis framework being used. For example, a reference could be a value in an SSA graph or something more precise for a context-sensitive data-flow graph. Further, a class could be a declared class, an object contour (see Section 4.2), or just the results of a specific set of `new` statements.

Figure 6 shows this information for the `lower_right` field of the `Rectangle` object in our example. The table labels values with the contours shown in Figure 5(b). All values that contain objects assigned to or read from `lower_left` in E_{lower_right} and all values that hold `Rectangles` from the creation of `r` are in R_{lower_right}. In Figure 6, the results for these sets are *precise*; the variables in E_{lower_right} are only used as the `lower_left` field of a `Rectangle`. If this were not the case – i.e. if some member of E_{lower_right} only *might* be from a `Rectangle` – there would be no way to generate a single sequence of code for the field reference.

$$
\begin{array}{rcl}
E_{lower\_right} & \leftarrow & \{this_4, q_0, lr_2, this_3\} \\
R_{lower\_right} & \leftarrow & \{r_0, this_2\} \\
Up_{lower\_right} & \leftarrow & \left\{ \begin{array}{lcl} this_4 & \rightarrow & r_0 \\ q_0 & \rightarrow & r_0 \\ lr_2 & \rightarrow & this_2 \\ this_3 & \rightarrow & this_2 \end{array} \right\}
\end{array}
$$

Figure 6: Example R, E and Up

For a field to be inlinable, not only must R_f and E_f be precise, but Up_f must be a realizable function. This is because object inlining requires Up_f to direct a program transformation – substituting container values for containee values – and realizability ensures that container values need not be used in places where they do not exist (e.g. before they are created). For example, in Figure 6, Up_f maps $this_4$ to r_0, but the two variables are in different scopes, so it must be possible to pass r_0 into the constructor (or alternately inline the constructor) in order to replace $this_4$ with it.

5.2 Analysis Frameworks

Analyses to derive inlinable fields can use any data-flow analysis framework, but the choice critically affects cost (space and time) and effectiveness (inlinable fields identified). We explore three different analysis frameworks, ranging from local data-flow analysis to adaptive flow-sensitive analysis [24]. These frameworks allow us to explore the cost-effectiveness space of inlining analyses.

Local Data Flow Conventional intra-procedural analysis which is fast, but limits the identification of inlinable fields to those for which both E_f and R_f confined to a single procedure. This limitation could be mitigated by good procedure inlining heuristics.

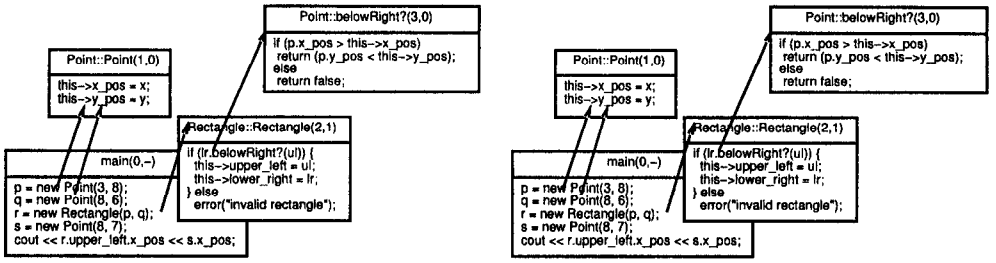


Figure 5: Adaptive Flow Analysis

nCFA A conventional form of inter-procedural analysis with flow-sensitivity based on n of levels of calling context [28]. Because of compute cost exponential in n, only small values of n are practical. The context sensitivity of nCFA allows inlining of objects for which E_f and R_f span procedure boundaries.

Adaptive Analysis (described in Section 4.2) An inherently whole-program analysis with demand-driven – and arbitrarily deep – context sensitivity. It is capable of distinguishing E_f and R_f thru arbitrarily many levels of procedure calls. A significant advantage of adaptive analysis over nCFA is that data flow is tracked even thru assignments to other object fields via *object contours*.

5.3 Local Data Flow

The local data flow analysis uses a single mechanism, *tagging*, to compute E_f, R_f and Up_f simultaneously for each field in a given method. Local data flow works on individual methods, for which the Concert Compiler uses SSA and the PDG (see Section 4.1) and values are *tagged* with information about object fields to which they are assigned or from which they are read. We first define precisely what a *tag* is, then discuss how they are propagated by local data flow, and finally show how to compute E_f, R_f and Up_F from them.

5.3.1 Tags

A tag is associated with a given program value; it is a sequence $\langle \langle object \otimes field \rangle_1, \ldots, \langle object \otimes field \rangle_n \rangle$ indicating the object fields to which that value is assigned or from which it is read. The *objects* are the SSA values representing the value from which the field was accessed and *fields* are the fields accessed. For instance, `a = b.f` would yield a tag of $\langle \langle \mathtt{b} \otimes \mathtt{f} \rangle \rangle$ for `a`. Manipulation of tags is defined as follows:

$$
\begin{aligned}
NoTag &\implies \text{value not from any field} \\
Object(\langle o \otimes f \rangle) &\implies o \\
Field(\langle o \otimes f \rangle) &\implies f \\
Head(\langle \langle o \otimes f \rangle_1, \ldots, \langle o \otimes f \rangle_n \rangle) &\implies \\
&\quad \langle o \otimes f \rangle_1 \\
MakeTag(\langle o \otimes f \rangle, \langle \langle o \otimes f \rangle_1, \ldots, \langle o \otimes f \rangle_n \rangle) &\implies \\
&\quad \langle \langle o \otimes f \rangle, \langle o \otimes f \rangle_1, \ldots, \langle o \otimes f \rangle_n \rangle
\end{aligned}
$$

5.3.2 Tag propagation

The tags defined above are propagated by a standard data-flow algorithm thru the program graph of an individual method in order to compute E_f, R_f and Up_f for all fields f of all objects created in that method. In defining our data-flow equations, we will use the $\Rightarrow$ and $\Leftarrow$ relations to indicate forward and backward data-flow relationships between values, and $+$ to signify transitive closure.

We use two sets of tags for each value in the program graph, *ForwardTags* and *BackwardTags*, to record fields from which the value is read and to which it is assigned respectively. Tag propagation conceptually starts at the creation sites (i.e. `new` statements) of all uncontained objects, that is, objects that are not assigned into any field of any object. The set of uncontained objects, which we call *Top*, is defined as follows:

$$
Top \leftarrow \left\{ v \;\middle|\; \begin{array}{l} kind(creation(v)) = creation \wedge \\ \neg \exists_{v_i, n} \left(\begin{array}{c} kind(n) = access \wedge \\ n \in reaching(v_i) \wedge \\ v \Rightarrow^+ v_i \end{array} \right) \end{array} \right\}
$$

Propagation proceeds from the uncontained objects according to data-flow rules that tag all definitions and uses of object fields with tags corresponding to the appropriate object and field values. The rule for object creations is straightforward: for each creation site in *Top*, its result is given the forward tag NoTag:

$$
\mathtt{v = new\ Obj} \wedge v \in Top \implies ForwardTags(v) \leftarrow \{NoTag\}
$$

Local data flow must treat field access nodes specially, as fields represent global state rather than SSA values; there are two rules for these field accesses – one for reads and one for writes – that propagate the appropriate tag based upon the field accessed and the container's tag:

$$
Tags(v) \leftarrow ForwardTags(v) \cup BackwardTags(v)
$$

$$
\mathtt{v = o.f} \implies ForwardTags(v) \leftarrow \{MakeTag(\langle o \otimes f \rangle, t_1) | t_1 \in Tags(o)\}
$$

$$\texttt{o.f = v} \Longrightarrow BackwardTags(v) \leftarrow \{MakeTag((o \otimes f), t_1) | t_1 \in Tags(o)\})$$

Finally, there are the rules for propagating tags across data-flow constraints:

$$ForwardTags(v) \leftarrow$$
$$ForwardTags(v) \cup \bigcup_{\{i|i \Rightarrow v\}} \{t \,|\, t \in ForwardTags(i)\}$$

$$BackwardTags(v) \leftarrow$$
$$BackwardTags(v) \cup \bigcup_{\{i|i \Leftarrow v\}} \{t \,|\, t \in BackwardTags(i)\}$$

5.3.3 Computing E_f, R_f and Up_f

Once the tag propagation is done, we compute E_f, R_f and Up_f for each field of each object created in the method. We will give the definitions in terms of an arbitrary field f of an object v where v is the result of an object creation (i.e. a **new** statement). R_f is the transitive closure, *forward and backward*, across the data flow from v throughout the method, and E_f is all the values tagged with a pair representing the field f and an object in R_f. Note that the tags have recorded the container value from which the field was extracted, which allows us to find the corresponding container for a given member of E_f by looking the the *Object* of its tag.

$$\begin{aligned}
R_f &\leftarrow \{v_i \,|\, v(\Leftarrow \cup \Rightarrow)^+ v_i\} \\
E_f &\leftarrow \left\{ v_i \,\middle|\, \exists_{t \in Tags(v_i)} \begin{pmatrix} Field(Head(t)) = f \wedge \\ Object(Head(t)) \in R_f \end{pmatrix} \right\} \\
Up_f &\leftarrow \left\{ (e, r) \,\middle|\, \begin{matrix} e \in E_f \wedge \\ \exists_{t \in Tags(e)} Object(Head(t)) = r \end{matrix} \right\}
\end{aligned}$$

There are no criteria for the precision of R_f for it is defined rather than calculated; however, both E_f and Up_f must be checked for ambiguity. If a value in E_f has more than one tag, then that value could be from multiple containers, making inlining invalid. Similarly, Up_f must be a many-to-one mapping, so a given member of E_f must not be mapped to more than one member of R_f:

$$\begin{aligned}
E_f &: \neg\exists_{v \in E_f} \left\{ \begin{matrix} |ForwardTags(v)| > 1 \vee \\ |BackwardTags(v)| > 1 \end{matrix} \right\} \\
Up_f &: \neg\exists_{e, r_1, r_2} (e, r_1) \in Up_f \wedge (e, r_2) \in Up_f
\end{aligned}$$

5.3.4 Realizability

We mentioned in Section 5.1 that the mapping Up_f must be checked for realizability: we must ensure that the specified transformation is legal. Recall that the transformation is simply substituting container values for uses of containee values according to Up_f. Values with a *ForwardTag* are no problem, for the corresponding container must exist; for values with a *BackwardTag*, we are pushing the corresponding container backward along data-flow paths to control conditions where it may not exist. Since we are using the PDG (see Section 4.1), the control conditions can be verified simply: for each mapping (e, r) where e has a *BackwardTag*, check that the creation of e is below the creation of r in the PDG and the node in r's region that controls e (or is e) is not constrained to happen before the node creating r. These constraints are formalized in Figure 7

5.4 nCFA Analysis

The nCFA analysis, just like the local one, uses *tagging* to compute E_f, R_f; then we will use the resultant E_f, R_f and inter-procedural data-flow graph to construct Up_f. Tagging is used to identify uses of different containee objects, relying upon the statically created contours to provide needed context sensitivity. We first define precisely what a *tag* is, then discuss how they are propagated using nCFA analysis. Finally we detail how to compute E_f, R_f and Up_f.

This analysis—and the similar adaptive analysis discussed next—is based upon techniques we devised previously [9]. The tag propagation for forward tags is exactly the same, but our prior technique worked by copying fields of the child object into the fused object, which we called definition specialization. That proved inadequate when evaluated on larger programs, so we replaced that mechanism with backward tags.

5.4.1 Tags

A tag is a sequence $\langle field_1, \ldots, field_n \rangle$ indicating from or to which fields a given value is read or assigned. The *field*s are field names from the program's classes. Manipulation of tags is defined as follows:

$$\begin{aligned}
NoTag &\Longrightarrow \text{not from field.} \\
MakeTag(f, \langle f_1, \ldots, f_n \rangle) &\Longrightarrow \langle f, f_1, \ldots, f_n \rangle \\
Head(\langle f_1, \ldots, f_n \rangle) &\Longrightarrow f_1
\end{aligned}$$

5.4.2 Tag propagation

The tags defined above are propagated by a standard data-flow algorithm thru the inter-procedural data-flow graph in order to compute E_f, R_f. In defining our data-flow equations, we will use the $\Rightarrow$ and $\Leftarrow$ relations to indicate forward and backward data-flow relationships between values.

Tag propagation conceptually starts at the creation sites (i.e. **new** statements) of classes, that is, classes values of which type are not assigned into any field of any object. The set of uncontained classes, which we call *Top*, is defined as follows:

$$Top \leftarrow \{c \,|\, \neg\exists_{C \in AllClasses} \exists_{F \in Fields(C)} c \in Creators(F)\}$$

Tags are propagated thru the inter-procedural data-flow graph along forward and back data-flow paths, with special rules for the results of object creations and field accesses. The rule for object creations is straightforward: for each creation site in *Top*, its result is given

$$
\begin{aligned}
RegionAbove(r_1, r_2) &\leftarrow r_1 = r_2 \vee (parent(r_2) \wedge RegionAbove(r_1, region(parent(r_2)))) \\
Above(n_1, n_2) &\leftarrow RegionAbove(region(n_1), region(n_2)) \\
After(n_1, n_2) &\leftarrow n_1 \succ n_2 \vee After(n_1, parent(region(n_2))) \vee After(parent(region(n_1)), n_2) \\
LocalDominates(n_1, n_2) &\leftarrow Above(n_1, n_2) \wedge \neg After(n_1, n_2) \\
Realizable(Up_f) &\leftarrow \forall_{(e,r)\in Up_f} LocalDominates(creation(r), creation(e))
\end{aligned}
$$

Figure 7: Realizability

the forward tag *NoTag*:

$$
\texttt{v = new Class} \wedge Class \in Top \Longrightarrow \\
ForwardTags(v_c) \leftarrow \{NoTag\}
$$

There are two rules for field accesses – one for reads and one for writes – that propagate the appropriate tag based upon the field accessed and the container's tag (in the subsequent equations, recall that v_c is v specialized to contour c):

$$
\texttt{v = o.f} \Longrightarrow ForwardTags(v_c) \leftarrow \\
\bigcup_{t\in Tags(o_c)} \left\{ \bigcup_{c\in Creators(o_c)} MakeTag(f,t) \right\}
$$

$$
\texttt{o.f = v} \Longrightarrow BackwardTags(v_c) \leftarrow \\
\bigcup_{t\in Tags(o_c)} \left\{ \bigcup_{c\in Creators(o_c)} MakeTag(f,t) \right\}
$$

Finally, there are the rules for propagating tags across data-flow constraints (the restrictive clauses prevent extraneous propagation of tags across dynamic dispatches):

$$
ForwardTags(v_c) \leftarrow \\
\bigcup_{\{x | x \Rightarrow v_c\}} \left\{ t \,\middle|\, \begin{array}{l} t \in ForwardTags(x) \wedge \\ Creators(Head(t)) \cap Creators(v_c) \end{array} \right\}
$$

$$
BackwardTags(v_c) \leftarrow \\
\bigcup_{\{x | v_c \Rightarrow x\}} \left\{ t \,\middle|\, \begin{array}{l} t \in BackwardTags(x) \wedge \\ Creators(Head(t)) \cap Creators(v_c) \end{array} \right\}
$$

5.4.3 Computing E_f, R_f and Up_f

E_f and R_f are each defined in terms of a class. R_f is values flowing from the creations of the class containing f, and E_f is values flowing from the creations of the classes of the child (i.e. the type of f):

$$
\begin{aligned}
Container(f) &\leftarrow \text{the class containing field f} \\
CreationPoints(C) &\leftarrow \text{results of } \texttt{new}\text{s of C}
\end{aligned}
$$

$$
E_f \longrightarrow \left\{ v \,\middle|\, \bigcup_{c\in CreationPoints(Container(f))} c \Rightarrow^+ v \right\}
$$

$$
R_f \longrightarrow \left\{ v \,\middle|\, \bigcup_{C\in Creators(f)} \bigcup_{c\in CreationPoints(C)} c \Rightarrow^+ v \right\}
$$

Computing Up_f requires mapping from a given containee value from E_f to the appropriate container value in R_f, which involves finding the value to or from which the containee goes or comes. We tackle finding the value differently for forward and backward tagged values. Forward tagged values flowed from a container (hence the tag) and that is the one to use in place of it. At the points where the containee is extracted (i.e. at field reads), we have an association between the container and containee values, and so we can use the container value instead. This gives us Up_f for forward tagged values:

$$
e \in E_f \wedge ForwardTag(e) \Longrightarrow \\
\texttt{o = r.f} \wedge o \Rightarrow^+ e \rightarrow (e, r) \in Up_f
$$

Unhappily, backward tagged values are more complex. There is the same association at points where the containee is inserted (i.e. at field writes), and that does tell us which container object to use. However, in order to use this container value, we must pass it backwards along the data-flow paths for the containee values, which may not be possible. We deal with this by trying to find some value in R_f that *dominates* (denoted $\equiv$) both the place where the containee is assigned (so that we are using the right container) and the creation of the containee (so that we can replace the right containee). We need a very strict notion of dominance: $i \equiv j$ means the statement creating of j executes *at most once* for each execution of the statement creating i.

$$
(e, r) \in Up_f \Longrightarrow \\
\texttt{o.f = e} \wedge \\
r \equiv^+ o \wedge \\
\forall_{C\in Creators(e)} \forall_{c\in CreationPoints(C)} r \equiv^+ c
$$

$$
i \equiv j \leftarrow \\
(i \succ j \wedge kind(j) = argument) \vee \\
LocalDominates(creation(i), creation(j))
$$

5.5 Adaptive Analysis

The adaptive flow analysis (see Section 4.2) uses *tagging* and adaptive splitting together to compute E_f, R_f; then we will use the resultant E_f, R_f and inter-procedural data-flow graph to construct Up_f. Tagging and adaptive method splitting is used to disambiguate uses of different containee objects, and adaptive object splitting creates individual object contours representing the creations of containee objects. It is similar to the nCFA analysis, but is defined on object contours rather than classes, and creates object contours as needed. We first define precisely what a *tag* is, then discuss how they are propagated using adaptive analysis. Finally we detail how to compute E_f, R_f and Up_f.

5.5.1 Tags

A tag is a sequence $\langle field_{countor_1}, \ldots, field_{contour_n} \rangle$ indicating from or to which fields of which contours (see Section 4.2) a given value comes or goes: *contours* are the object contours representing the creator of the object accessed and *fields* are field names from their respective contours' classes. Manipulation of tags is defined as follows:

$$\begin{aligned} NoTag &\implies \text{not from field.} \\ MakeTag(f_c, \langle f_{c_1}, \ldots, f_{c_n} \rangle) &\implies \langle f_c, f_{c_1}, \ldots, f_{c_n} \rangle \\ Head(\langle f_{c_1}, \ldots, f_{c_n} \rangle) &\implies f_{c_1} \end{aligned}$$

5.5.2 Tag propagation

The tags defined above are propagated by a standard data-flow algorithm thru the inter-procedural data-flow graph in order to compute E_f, R_f. Adaptive analysis is an iterative algorithm, and this tag propagation is repeated for each iteration of the analysis framework. In defining our data-flow equations, we will use the $\Rightarrow$ and $\Leftarrow$ relations to indicate forward and backward data-flow relationships between values.

Tag propagation conceptually starts at the creation sites (i.e. `new` statements) summarized by uncontained contours, that is, objects that are not assigned into any field of any object. The set of uncontained contours, which we call *Top*, is defined as follows:

$$Top \leftarrow \left\{ c \;\middle|\; \begin{array}{l} \neg \exists_{C \in AllClasses} \\ \quad \exists_{F \in Fields(C)} \\ \quad\quad \exists_{c_1 \in Creators(C)} c \in Creators(F_{c_1}) \end{array} \right\}$$

Tags are propagated thru the inter-procedural data-flow graph along forward and back data-flow paths, with special rules for the results of object creations and field accesses. The rule for object creations is straightforward: for each creation site in Top, its result is given the forward tag NoTag:

$$\texttt{v = new Obj} \wedge (Creators(v_c) - Top) = \emptyset \implies ForwardTags(v_c) \leftarrow \{NoTag\}$$

There are two rules for field accesses – one for reads and one for writes – that propagate the appropriate tag based upon the field accessed and the container's tag:

$$\texttt{v = o.f} \implies ForwardTags(v_c) \leftarrow \bigcup_{t \in Tags(o_c)} \left\{ \bigcup_{c \in Creators(o_c)} MakeTag(f_{self(c)}, t) \right\}$$

$$\texttt{o.f = v} \implies BackwardTags(v_c) \leftarrow \bigcup_{t \in Tags(o_c)} \left\{ \bigcup_{c \in Creators(o_c)} MakeTag(f_{self(c)}, t) \right\}$$

Finally, the rules for propagating tags across data-flow constraints are exactly the same as for nCFA.

5.5.3 Adaptive splitting

The propagated tags are used to guide adaptive splitting to disambiguate the uses and creations of different containee objects. The analysis framework allows individual analyses to register discriminator functions that determine whether a given contour needs to be split. Uses of different containees are disambiguated using the forward tags by splitting contours of methods in which differing tags from different callers merge. Each contour has a set of edges representing the calls it summarizes. Contours are split so that all incoming edges have compatible tags. The edges of a contour are partitioned into sets that become new contours as shown in Equation (1) of Figure 8; note that $Arg(e, i)$ is the ith argument of edge e.

The object contours representing the creation of containee objects are created by using the backward tags. These backward tags will be propagated from assignments to fields back toward the creations of the objects being assigned into that field, which must ultimately be object creation statements, which are represented by object contours. Two steps are involved in splitting object contours: first, the data-flow path from the field back to the creation must be separated by splitting all intermediate methods according to the backward tags of their values as given in the formula. In Equation (2) of Figure 8 $Ret(e, i)$ is the ith return value of edge e.

Then the object contours themselves must be split. The object contours represent the result of a `new` operation, so the backward tags will ultimately flow all the way back to that value, and so these tags can be used to partition the object contour into a set of contours. This is shown in Equation 3 of Figure 8; the result of the `new` statement is v in that equation.

5.5.4 Computing E_f, R_f and Up_f

In the adaptive analysis, an object contour is created to represent the containee objects, so E_f and R_f are each defined in terms of an object contour. A field f is discriminated by the object contour of its container, which we designate f_c. The computation is exactly the same as for nCFA, except that fields are specialized to

$$\{\{e_1,\ldots,e_n\}\,|\neg\exists_{i,j,k} ForwardTags(Arg(e_i,k)) \neq ForwardTags(Arg(e_j,k))\} \quad (1)$$

$$\{\{e_1,\ldots,e_n\}\,|\neg\exists_{i,j,k} BackwardTags(Ret(e_i,k)) \neq BackwardTags(Ret(e_j,k))\} \quad (2)$$

$$\{\{c_1,\ldots,c_n\}\,|\neg\exists_{i,j,v} BackwardTags(v_{c_i}) \neq BackwardTags(v_{c_j})\} \quad (3)$$

Figure 8: Criteria for Adaptive Splitting

object contours. So, R_f and E_f for a field f_o become

$$CreationPoints(c) \leftarrow \text{results of } \texttt{news} \text{ of } c$$

$$E_f \longrightarrow \left\{ v \,\middle|\, \bigcup_{c \in CreationPoints(o)} c \Rightarrow^+ v \right\}$$

$$R_f \longrightarrow \left\{ v \,\middle|\, \bigcup_{C \in Creators(f_o)} \bigcup_{c \in CreationPoints(C)} c \Rightarrow^+ v \right\}$$

Computing Up_f is exactly the same as for nCFA, except that it is done for the E_f and R_f specialized for a given contour.

5.6 Object Inlining Transformation

Once we have E_f, R_f and Up_f for a given field f, the object inlining transformation is simplicity itself, at least conceptually: replace uses e of the containee with $Up_f(e)$, move the containee's storage into the container, and delete all creations of a containee[5].

6 Evaluation

We set out to determine how much analysis power is required for effective automatic inline object allocation on a range of programs. Thus, we must measure how effective and costly each analysis option is at compile time, and their effects at runtime. Our primary metrics measure the compile-time benefits and costs – inlined field counts and analysis costs – and dynamic runtime changes, field accesses and object allocations eliminated. We also measure the impact of object inlining on the program overall. First, we describe what compilers (Section 6.1), benchmarks (Section 6.2) and metrics (Section 3) we used. We present our results in Section 6.4.

6.1 Methodology

Our evaluation uses a range of C++ programs to compare our three inlining analyses – local, 1-cfa and adaptive – with a base program compiled by the Concert compiler with no object inlining. For calibration, we compare the runtime[6] and code size these four program versions with the same programs compiled with the latest gcc (2.8.1). Both compilers are run with full optimization (O3 for g++) and given the whole program. Thus, we use the five versions of each program compiled as shown in Figure 1. In this table, *analysis* is the object inlining analysis used (1-CFA is the traditional Control Flow Analysis), and *policy* whether inlining was done by hand, automatically or not at all. For the 1-CFA and adaptive analyses, analysis was performed before any transformation had taken place; in order to make the local scheme as effective as possible, we performed the local analysis and transformation *after* method inlining.

name	object inlining analysis	object inlining policy	compiler
base	none	none	Concert
local	local	automatic	Concert
1-cfa	1-CFA	automatic	Concert
adaptive	adaptive	automatic	Concert
c++	N.A.	manual	G++ 2.8.1

Table 1: Compared Analysis and Compiler Parameters

Since the benchmark programs are in C++, they include a lot of low-level information – e.g. specifying virtual versus non-virtual functions, and denoting register, stack, heap and inlined storage allocation – that is not compatible with the high-level model expected by the Concert compiler. The Concert compiler discards all such information[7], ignores type information and uses a reference model for all objects.

6.2 Benchmarks

We evaluate our object inlining techniques on the wide range of commonly-used standard data structures contained in two class libraries: the National Institutes of Health Class Library (NIHCL) and the Object Abstract Type Hierarchy (OATH) libraries. We use them as they are freely available and comprehensive. Both libraries come with a range of test programs that exercise the library, and we evaluate object inlining both on these test codes and also third-party programs written using the libraries. The NIHCL codes include the 20,000 line NIHCL library, and the OATH codes include the 18,000 line OATH library. In addition, we use some benchmark programs commonly used to evaluate object oriented systems; these programs exhibit a variety of data and control structures. These codes – and the inlining opportunities they exhibit – are summarized in Figure 2. The lines column has the lines of code in the

[5] Having E_f allows us to precisely find all creations because the results of such creations will be in E_f.

[6] All runs are done on a 266MHz Pentium Pro system with 128M of memory.

[7] Obviously, the generated code respects the *semantics* of different storage allocations, such as the meaning of assignment

program followed by the lines of library code in parentheses. These codes represent a superset of the benchmarks used in our prior work [9]; that work used only *silo*, *richards*, *polyover* and *oopack*.

6.3 Metrics

To assess the respective benefits and costs of our approaches to object inlining, we measure compile-time cost and effectiveness and the runtime impact. Our whole-program compiler makes no distinction between a program and the libraries it uses, so all metrics cover the program and all libraries together. Our metrics are summarized in Table 3 and described below.

Compile-Time Effectiveness We use a count of the total number of fields found to be inlinable. To calibrate the effectiveness of our techniques, we compare these counts with the number of fields manually declared inline.[8]

Compile-Time Cost We use the amount of analysis precision required, which we measure as the amount of context-sensitivity required per method and per class. For the adaptive analysis, these counts are the total number of contours created across all iterations.

Direct Dynamic Effects By fusing objects, inline allocation should reduce the number of field accesses and object allocations during program execution, and we measure these as our primary metric for runtime effectiveness.

Overall Dynamic Effects To assess the impact of object inlining on the program overall, we measure runtime, executable size and memory usage. The specialization required by inlining could increase code size, and reducing the number of objects allocated should reduce total memory usage.

6.4 Results

We present our metrics in same four groups as for Table 3: compile-time effectiveness, compile-time cost, direct dynamic effects and overall dynamic effects.

6.4.1 Compile-Time Effectiveness

Figure 9(a) presents the counts of inlinable fields discovered for each analysis and program; polymorphic fields are counted as a fraction based on how many of the polymorphic uses were found to be inlinable. The counts range from 0 in some cases for 1-cfa and local analyses to 9 for *stack* and *options* for the adaptive analysis. These are raw counts, so the trend is for more fields to be inlinable in larger programs as they typically have more classes.

[8]It would be nice to calibrate by determining the numbers of fields "really" inlinable, but this is problematic. One cannot determine this automatically – our automatic determination is what we are evaluating – and counting by hand is difficult and error-prone for codes the size and complexity of NIHCL and OATH.

Adaptive analysis proved most effective, finding at least as many fields as any other analysis, and a superset of those declared inline in C++. The 1-cfa analysis was as effective as adaptive analysis on some programs, but dramatically less so for others. It sometimes did not find all the fields declared inline in C++. The local analysis was ineffective: it found no fields inlinable on several programs, and it never found more than two fields inlinable.

Adaptive analysis – and sometimes 1-cfa as well – found fields not declared inline in C++ when there were objects conceptually in a dynamic relationship – such as cons cells – that were used statically in a given context. This demonstrates an advantage of automatic object inlining: the ability to discover fortuitously static uses of normally dynamic structures. The most common case was fusing a cons cell with its associated data in situation where analysis determined there was no sharing amongst the list elements. This happened for *options*, *pdl2a*, *silo* and *polyover*. Adaptive analysis could fuse polymorphic lists in *options*.

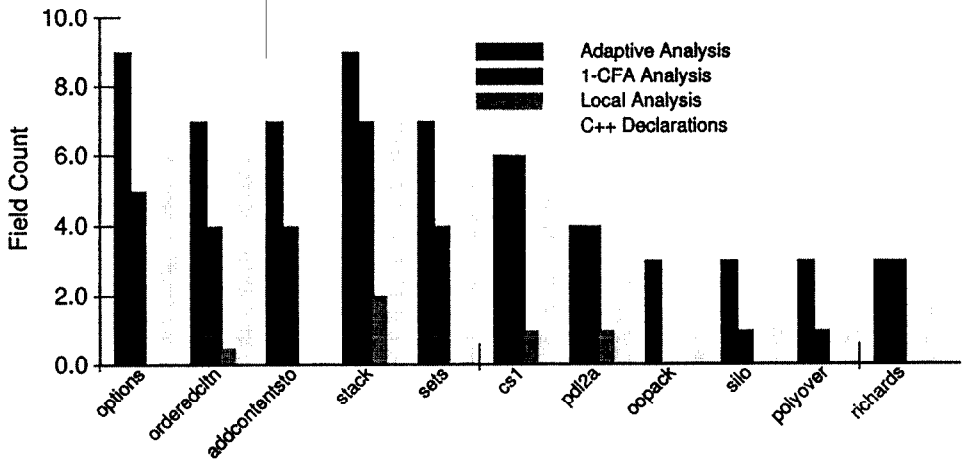
The 1-cfa analysis, lacking context sensitivity based upon object values, generally could not handle polymorphic structures; and, in each of *sets*, *addcontentsto* and *ordercltn*, the sensitivity limits of 1-CFA prevented it from specializing two fields actually declared inline in C++. Aside from those six fields, all other differences between adaptive and 1-cfa analysis are attributable to data sensitivity. The local data-flow analysis found very few fields, primarily because most of the inlinable fields were in core program data structures used throughout the program and hence not amenable to local analysis techniques. For *oopack*, the local analysis had trouble because all major data structures are passed thru global variables.

For several of the NIHCL benchmarks, one array in the core data structures could not be inlined because it may be reallocated dynamically to resize it if it overflows. This never actually happens in some of the benchmarks, but our compiler does not do sufficient range propagation to figure this out.

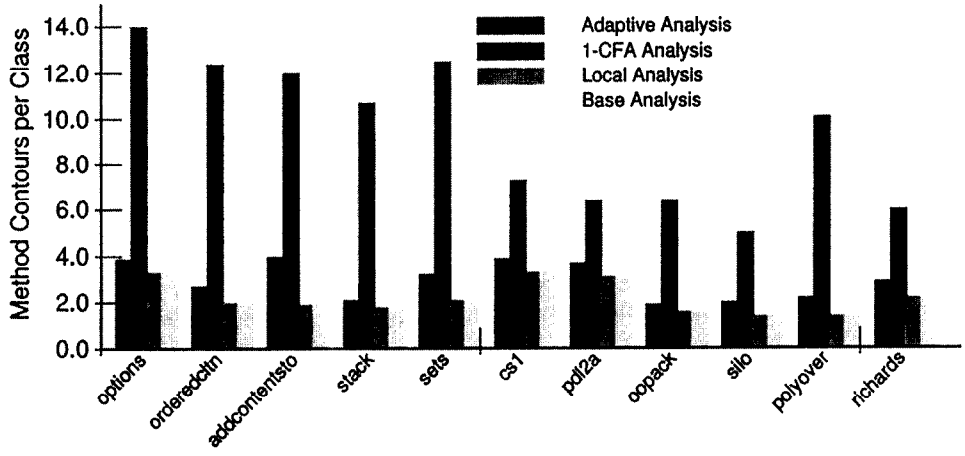
6.4.2 Compile-Time Cost.

The cost of our analyses is measured by the numbers of contours per method and class, as shown in Figures 9(b) and (c). The general trends for method contours are different for the different analyses. Adaptive analysis has fairly flat costs – between 2 and 4 contours per method – across the programs, whereas 1-cfa cost rises as program size grows, peaking for our largest benchmark, *options*. Local and base both use adaptive analysis without the object inlining component, and they both show flat costs of between 1.5 and 3.5 contours per method. A notable features of this graph is that adaptive object-inlining analysis does not raise the cost much over the base adaptive analysis, except on *addcontentsto* for which cost doubles.

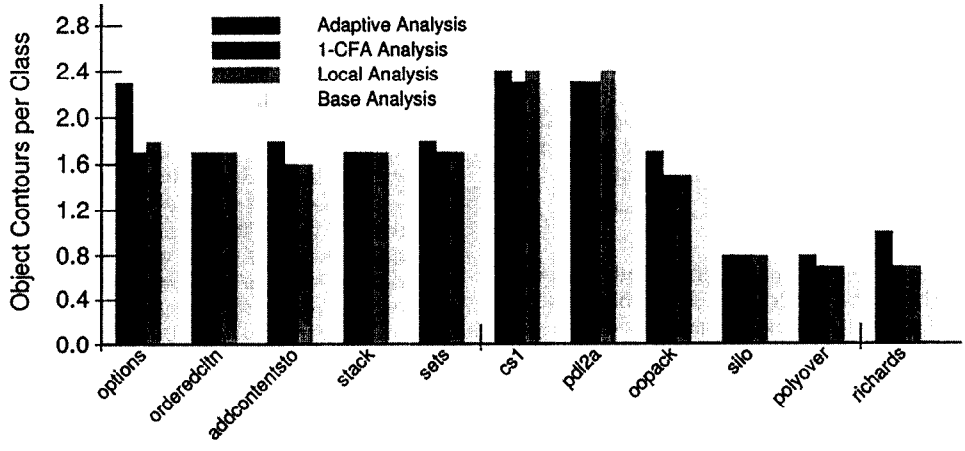
The rise in cost of 1-cfa for larger programs is expected, and has been observed before; the small variations in cost for adaptive analysis track the structural complexity of the programs. More polymorphic codes such as *richards* and *options* – which each use a polymorphic core data structure – have higher costs than *stack*,



(a) Inlinable Fields



(b) Contours Per Method



(c) Contours Per Class

Figure 9: Compile Measurements (absolute counts)

Program	≈ *lines*	*main data structures*	*inlinable objects*
		NIHCL	
options	30K	polymorphic lists	list elements, empty, 1,2d-arrays
orderedcltn	20.11K	dynamic arrays	iterators, collection wrappers, arrays
stack	20K	stacks	parent/child, iterators, wrappers, arrays
addcontentsto	20K	sets, dynamic arrays	iterators, wrappers, arrays
		OATH	
cs1	18.1K	arrays, smart pointers	wrappers, parent/child, smart pointers
pdl2a	18.1K	queues, smart pointers	wrappers, list elements
		other programs	
oopack	3.3K	arrays, numbers	array of objects
silo	1.3K	queues	conses, list wrappers
polyover	1.2K	lists, objects	array of objects, conses
richards	1.0K	objects	simple parent/child

Table 2: Benchmark Summary

Name	Units	Description
Compile time effectiveness		
inlinable fields	count	number of inlined fields
Compile time cost		
method contours	per method	method contours generated
object contours	per class	object contours generated
Direct dynamic effects		
reads	count relative to base	number of object field reads
news	count relative to base	number of object allocations
Overall dynamic effects		
runtime	time relative to base	program runtime
memory usage	bytes relative to base	total heap allocation
code size	bytes relative to base	executable image size

Table 3: Metrics

silo and *oopack* which have no polymorphism whatsoever. An anomaly is the relatively small difference between adaptive and 1-cfa on the *OATH* codes *cs1* and *pdl2a*. These codes use an idiom for nil that creates type ambiguities throughout the program, which causes substantial demand-driven splitting in an attempt to resolve them.

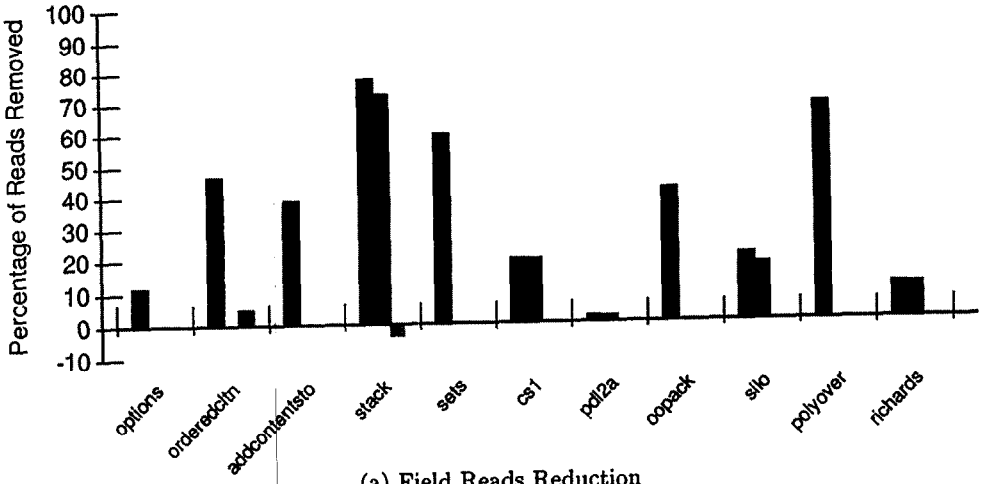
Figure 9(c) shows that few object contours are needed in general – none of the bars go much above two. The costs are fairly constant between analyses, except for some peaks for adaptive analysis. These peeks represent the extra precision needed for the most polymorphic codes, particularly *options* and *richards*. This extra precision is what allows adaptive analysis to inline polymorphic fields. There are two anomalies. The first is that some bars are below one, which happens because some classes – such as abstract base classes – are not instantiated at all and so do not generated object contours. Even though 1-cfa does not create object contours, some of its numbers are still greater than one due to splitting of the array class required due to implementation artifacts.

The actual runtime of our analysis system varies about two minutes on the simplest program (*oopack*) without object inlining analysis to about 30 minutes for object inlining analysis on the largest code (*options*). This varies from one quarter to one half of the total compile time depending upon how much adaptive analysis is required. Asymptotically, the runtime varies both in accordance with program size and with the amount of demand-driven sensitivity required; however, the actual runtimes for individual programs are not shown since they are dominated by implementation details rather than directly by properties of the program[9].

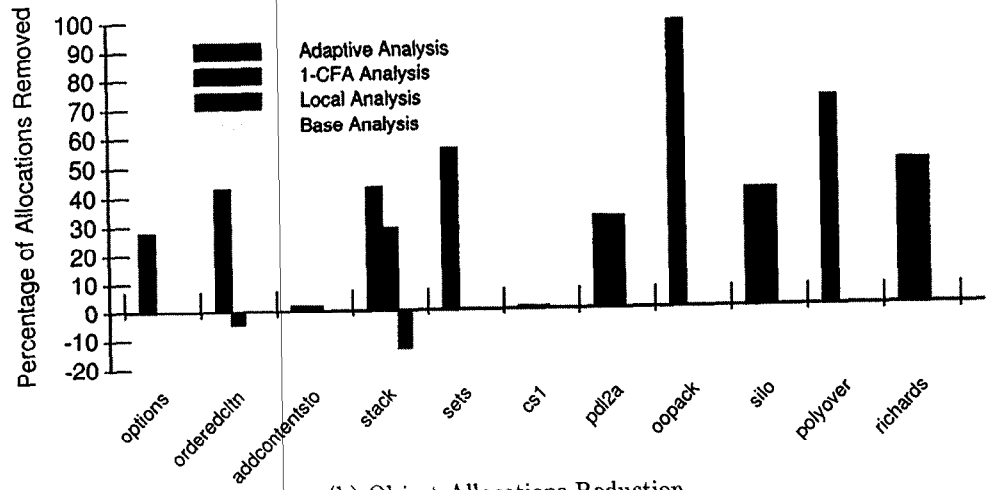
6.4.3 Direct Dynamic Effects

The charts in Figure 10(a) and (b) present details of the changes in object accesses and allocations induced by object inlining. They show the fraction of field reads removed and of object allocations removed respectively. The results are very varied for the adaptive and 1-cfa analyses; the peaks of both charts are high: for some programs, almost all allocation and references can be eliminated. On the other hand, some programs – such as *options* and the *OATH* codes show little gain on either metric. The average fraction of reads and allocations eliminated are 37% and 43% respectively over the base Concert program. The local analysis proved ineffective:

[9]For example, we use unsorted lists to record types, so computing a type difference is an $O(n^2)$ operation. Programs with lots of classes suffer unnecessarily from this artifact.



(a) Field Reads Reduction



(b) Object Allocations Reduction

Figure 10: Direct Runtime Measurements (relative to base)

only one program showed noticeable gains, due to how few inlinable fields the local analysis found.

There is some correlation with the inlinable field counts – as one might expect. The *stack* code – which tied the larger *options* for the most inlinable fields at 9 – shows dramatic gains on both metrics: about 80% of reads and 50% of objects both vanish. Large reductions of reads for *addcontentsto*, *oopack* and *polyover* with adaptive analysis are due to removal of object dereferences from critical paths in tight loops.

The most dramatic reduction of object allocation is for *oopack*: almost 100% are eliminated. One array of complex numbers accounts for the vast bulk of the objects in this benchmark, so inline allocating the arrays elements removes almost all the objects. The same is true to a lesser degree in *polyover*.

On the other hand, a couple of programs show little improvement in either metric even with adaptive analysis. The *options* code shows relatively modest gains despite having the most inlinable fields because it makes heavy use of strings, which implementation details prevent us from inline allocating. The *cs1* code – and the *oath* library in general – have relatively few inlinable fields due to type ambiguities caused by its idiom for nil objects.

6.4.4 Overall Dynamic Effects

Runtime. The relative execution times of our benchmarks are shown in Figure 11(a); this chart shows the fractional performance improvement relative to the base Concert code. We are evaluating the effects of object inlining relative to the base Concert code, as that is a controlled experiment simply turning object inlining on and off. Given the completely different implementations of g++ and Concert, the comparison with g++ is meant only as calibration of Concert's base performance. Compared with g++, the Concert compiler produces slower code on 6 of the benchmarks and faster code on 5 of them.

The chart show mostly performance gains up to 50% (for *polyover*) for object inlining using adaptive analysis. The average runtime gains are 3% for 1-cfa and 10% for adaptive analysis. The local analysis makes no appreciable difference on any code. The most significant performance gains are for *stack*, *oopack* and *polyover*. These gains come partially from the removed objects and reads, but are also due in large part to object inlining enabling other optimizations, especially caching fields in registers and allocating objects with provably limited lifetimes on the stack.

The scarcity of performance gains from object inlining – two programs are even slightly slower – on the other codes despite sometimes dramatic drops in read and object counts can be explained in part by our compiler. Our research focuses on high-level analysis and transformation, and we have a relatively simple code generator that is unambitious with local code optimizations[10]. Thus, when object inlining produces tighter bodies of code, that does not always translate into better performance.[11] However, object inlining dramtically reduces references and allocations, and making our backend take advantage of the better code is continuing work.

Memory Usage. Figure 11(b) shows the reduction in memory usage for the inlined program versions as a fraction of that used by the base Concert program. Both the adaptive and 1-cfa programs show significant reductions for many programs; the reduction is due to the reduced overhead of fewer objects in our garbage collected model and the space saved by the elided pointer fields. The average reduction in memory allocation is 3% for 1-cfa and 13% for adaptive analysis. In general, the correlation between reduced object allocations and reduced storage use is weak because of the variance in the size of single objects.

The greatest reduction occurs for the *stack* code with adaptive analysis: in this case, in addition to removing objects, object inlining enabled object state caching that allowed other objects to be pruned completely. Conversely, the increase in object allocation caused by the local analysis on *richards* is caused by inlining inhibiting other code optimizations, particularly object stack allocation.

Code Size. Figure 11 shows the size of the final program relative to the base executable produced by Concert. The various version produced by the Concert compiler are of almost identical size for each program, showing the specialization required by inline allocation does not result in significant code expansion. This is because it by and large the specialized methods would have to copied by inlining anyway. Compared with G++, the Concert compiler produces smaller executables for the large programs because it does a better job of tree-shaking the class libraries.

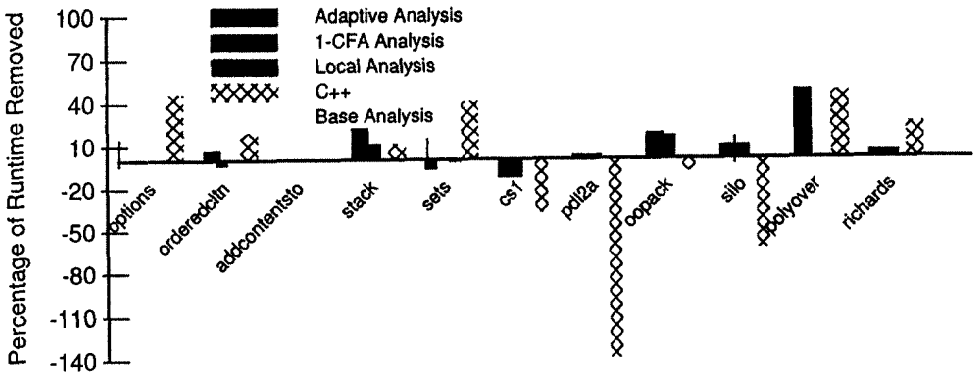
7 Related Work

Related work falls into two broad categories: there are other mechanisms for affecting inline allocation of objects, and there are other analyses that function similarly to a given aspect of object inlining but serve a different purpose.

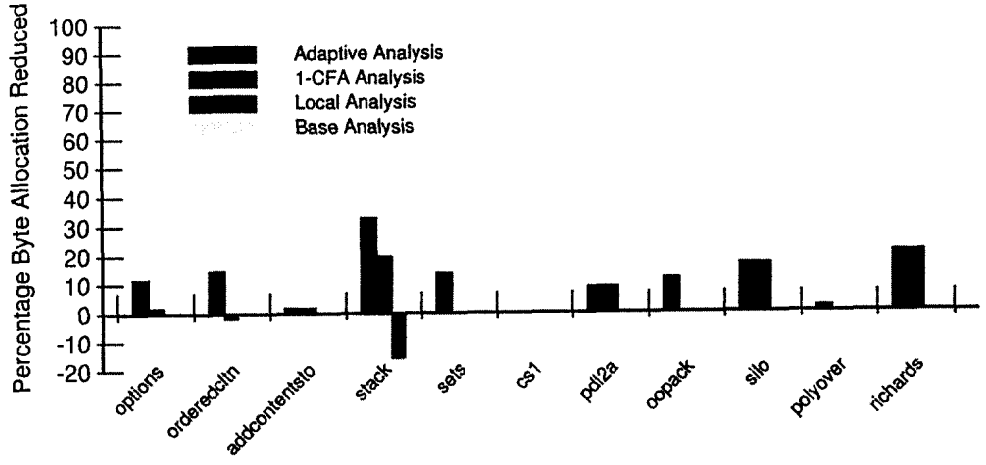
The idea of doing automatic object inlining dates back at least to the Emerald system, which has a reference object model [3] that was designed so that the compiler [18] could optimize object structures. However, while our adaptive analysis can produce the information needed for inline allocation (see Section 5.1), the simple, graph-algorithm-based analysis system of the Emerald compiler was sufficient only to allow the inlining of (boxed) immediate types. Immediate types in Emerald posed fewer analysis challenges for they had by definition the value semantics required for inlining. Budimlic and Kennedy [4] sketch a combined object and method inlining optimization which they call object inlining. In their scheme, for an object created within a method, all its called methods are inlined and the state of the object replaced with local variables. Our inter-procedural

[10] The register allocator is especially unhelpful on the Intel architecture with its scarcity of registers.

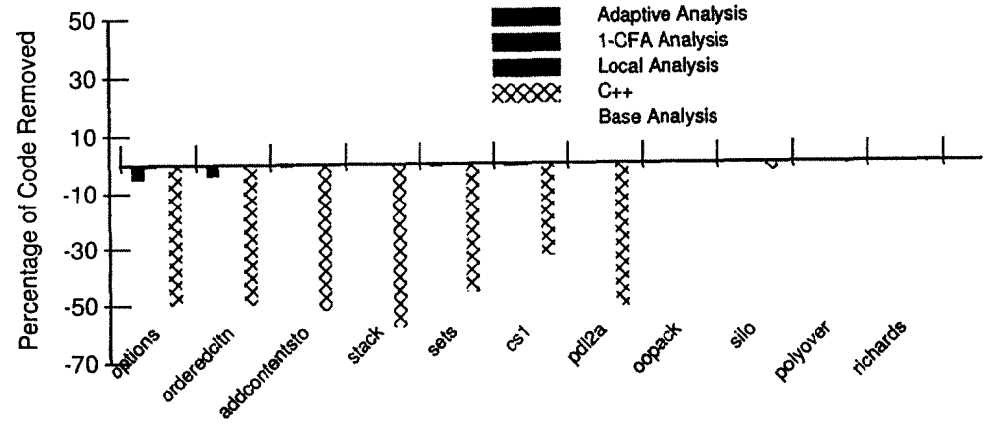
[11] This is also why Concert is slower than g++ on several of the benchmarks.



(a) Runtime Improvement



(b) Allocation Reduction



(c) Executable Size

Figure 11: Indirect Runtime Measures (relative to base)

analyses track field usage throughout the program – regardless of procedure boundaries – which was vital for inlining on our benchmarks; their scheme, like our local analysis, does not. But a detailed comparison is impossible as they give only a rough outline of their transformation which ignores the obvious aliasing concerns which we resolve by tagging.

Runtime optimizations analogous to object inlining have also been tried; witness cdr-coding as done in the Symbolics Lisp machines. The basic idea is that list elements are stored adjacently, eliminating the need for a tail pointer; this adjacency can be due to happenstance or can be arranged e.g. by a compacting garbage collector. Unlike our compile-time transformations, cdr-coding does not depend upon static analysis, and so can be applied to portions of lists and other entities to fine to be distinguishable by current static analysis techniques. On the other hand, our static techniques have no runtime overhead, whereas the fact that a cons cell is cdr-coded must be recorded and checked whenever the cell is accessed.

There has been much work in the functional community on *unboxing*, in which specialized representations are used to reduce storage and access overhead. Our adaptive flow analysis is able to compute precise inlining information in the presence of assignments to object fields; the unboxing work does not need to address this as there is no structure assignment in functional languages. The unboxing transformation of [20] handles polymorphism by generating specialized code only for monomorphic functions and coercing between general and unboxed representations as needed. On the other hand, our optimization is a global transformation that specializes polymorphic functions as needed.

In [15], Cordelia Hall and company present a transformation for Haskell that does generate specialized code to exploit unboxing for polymorphic functions. Their transformation resembles ours in that it propagates "unboxedness" throughout the program generating specialized code wherever needed. Our optimization is fully automatic and handles arbitrary user-defined object types. Due to the lazy semantics of Haskell, the transformation must be told what variables can be safely unboxed; furthermore, this transformation only unboxes immediate types.

In [26], Shao et al. unroll linked lists–essentially inline allocating tail pointers–in a functional subset of ML. Their analysis works using refinement types [13] that distinguish odd and even length lists. These refined types are propagated using an abstract interpretation, with rules for the refined types generated by `cons` statements. All functions that take list parameters are cloned and specialized with all possible combinations of refinement types for their list parameters. Our interprocedural analyses have two advantages. First, our field tags are more general, as they handle arbitrary object structures, rather than lists. Second, our interprocedural analysis analyzes only specializations that are actually used.

In [11], the authors describe access paths, which are used in various kinds of pointer analyses. The basic idea is that access paths keep track of object fields traversed during pointer dereferences. The major difference between access paths and our tags is that access paths start with stack variables, and are used for instance-based alias analysis, whereas our tags start from object creation sites and our analysis is class-based (actually object contour based). Object inlining analysis does not require the precision of instance-based aliasing, and so we can use a potentially cheaper class-based mechanism.

8 Summary

We have studied three compiler analyses to identify safely inlinable fields. These analyses span a range of cost and complexity, and all track field (member) accesses in heap objects. These analyses span a range of complexity from local data flow to adaptive whole-program, flow-sensitive inter-procedural analysis. Measuring the cost and effectiveness of these analyses on a suite of moderate-sized C++ programs (up to 30,000 lines including libraries), we find that object inlining optimizations eliminate 40% typically and as much as 90% of the object accesses and allocations, and can deliver significant performance benefits (averaging 10% faster but ranging from no improvement to 50%). But reaping these benefits requires powerful inter-procedural analysis that must focus effort to avoid excessive cost. Fortunately, the adaptive inter-procedural analysis we employed [24] computes precise information efficiently.

Acknowledgments

The Concert Compiler used for the experiments described in this paper has been the work of John Plevyak, Vijay Karamcheti, Xingbin Zhang and Hao-Hua Chu in addition to the present authors. In particular, the adaptive analysis techniques we use are the work of John Plevyak and Andrew Chien.

The research described in this paper is supported in part by DARPA orders #E313 and #E524 through the US Air Force Rome Laboratory Contracts F30602-96-1-0286 and F30602-97-2-0121, and NSF Young Investigator award CCR-94-57809. Support from Microsoft, Intel Corporation, Hewlett-Packard, and Tandem Computers is also gratefully acknowledged.

References

[1] O. Agesen, J. Palsberg, and M. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. In *Proceedings of ECOOP '93*, 1993.

[2] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of ECOOP*, pages 234–42. Springer-Verlag, June 1987.

[3] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the emerald system. In *Proceedings of OOPSLA '86*, pages 78–86. ACM, September 1986.

[4] Zoran Budimlic and Ken Kennedy. Optimizing java: Theory and practice. *Concurrency: Practice and Experience*, 9(6), June 1997.

[5] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying differences between C and C++ programs. Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.

[6] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 150–60, 1990.

[7] Andrew Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, and Xingbin Zhang. Supporting high level programming with high performance: The Illinois Concert system. In *Proceedings of the Second International Workshop on High-level Parallel Programming Models and Supportive Environments*, pages 15–24, April 1997.

[8] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programmin g Language Design and Implementation*, pages 93–102, La Jolla, CA, June 1995.

[9] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 7–17, Las Vegas, Nevada, June 1997.

[10] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[11] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, 1994.

[12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–49, July 1987.

[13] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1991.

[14] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++.* John Wiley and Sons, 1991.

[15] Cordelia Hall, Simon L. Peyton-Jones, and Patrick M. Sansom. *Functional Programming, Glasgow 1994*, chapter Unboxing Using Specialization. Workshops in Computing Science. Springer-Verlag, 1995.

[16] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 Conference Proceedings.* Springer-Verlag, 1991. Lecture Notes in Computer Science 512.

[17] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[18] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, University of Washington, Department of Computer Science, Seattle, Washington, 1987. TR-87-01-01.

[19] Christopher Lapkowski and Laurie Hendren. Extended ssa numbering: Introducing ssa properties to languages with multi-level pointers. In *Proceedings of CASCON*, 1996.

[20] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th Symposium on the Principles of Programming Languages*, pages 177–188, 1992.

[21] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA '91*, pages 146–61, 1991.

[22] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li. Thread scheduling for cache locality. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 60–71, 1996.

[23] John Plevyak. *Optimization of Object-Oriented and Concurrent Programs.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1996.

[24] John Plevyak and Andrew A. Chien. Precise concrete type inference of object-oriented programs. In *Proceedings of OOPSLA'94, Object-Oriented Programming Systems, Languages and Architectures*, pages 324–340, 1994.

[25] John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Proceedings of the Workshop for Languages and Compilers for Parallel Computing*, pages 566–580, 1995.

[26] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *ACM Conference on Lisp and Functional Programming*, June 1994.

[27] Olin Shivers. Control flow analysis in scheme. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–74. ACM, 1988.

[28] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University Department of Computer Science, Pittsburgh, PA, May 1991. also CMU-CS-91-145.

[29] Olin Shivers. *Topics in Advanced Language Implementation*, chapter Data-Flow Analysis and Type Recovery in Scheme, pages 47–88. MIT Press, Cambridge, MA, 1991.

[30] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, second edition, 1990.

[31] David Stoutamire and Stephen Omohundro. Sather 1.1, draft. Available online from `http://www.icsi.berkeley.edu/~sather/Sather-1.1.ps`, August 1995.

[32] Sun Microsystems Computer Corporation. *The Java Language Specification*, March 1995. Available at http://java.sun.com/1.0alpha2/doc/java-whitepaper.ps.

[33] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler.* Addison Wesley, 1992.

A Benchmark Programs

options is a command-line argument processing package. It's central data structure is a polymorphic list of command line options, in which different kinds of options—integer, real, string, etc—are represented by different subclasses of a generic `Option` class.

orderedcltn is a test program for the ordered collection classes of NIHCL; it uses sets and ordered collections, both of which in turn use expandable arrays; it also uses a variety of other NIHCL classes for handling I/O, iteration are other support functions.

addcontentsto is a test that creates a few ordered collections, and then loops adding 10,000 point objects to them. It uses sets and ordered collections, both of which in turn use expandable arrays; it also uses a variety of other NIHCL classes for handling I/O, iteration are other support functions.

stack tests the stack class of NIHCL. It creates stacks, ordered collections and other support objects, and pushes and pops objects of different classes (so the stacks are polymorphic) into them.

sets is another a test program for the ordered collection and set classes of NIHCL; it uses sets and ordered collections, both of which in turn use expandable arrays; it also uses a variety of other NIHCL classes for handling I/O, iteration are other support functions.

cs1 tests character sets, and creates lists, character objects and streams for doing I/O and inserting and deleting elements.

pdl2a tests doubly-linked lists. It creates lists, character objects and streams for doing I/O and inserting and deleting elements.

oopack is a set of tight numerical loops that use object extensively inside the loops. It uses iterator objects, matrix wrapper objects and complex number objects.

silo is a discrete event simulator benchmark. Its primary data structure is a list of events, which it uses as a queue. It has event objects, resource objects and various support objects.

polyover performs an overlay of two polygon maps. It uses lists and arrays of polygon objects to represent polygon maps.

richards is an operating system simulation benchmark; it uses a central task queue to which tasks are added in an event driven fashion when they receive messages.

B Raw Evaluation Results

	Analysis			
program	adaptive	1-cfa	local	c++
options	9	5	0	6
orderedcltn	7	4	1/2	6
addcontentsto	7	4	0	6
stack	9	7	2	8
sets	7	4	0	6
cs1	6	6	1	5
pdl2a	4	4	1	3
oopack	3	0	0	1
silo	3	1	0	2
polyover	3	1	0	2
richards	3	3	0	2

Table 4: Counts of Inlinable Fields

	Analysis				
program	adaptive	1-cfa	local	base	methods
options	1730	6154	1449	1449	441
orderedcltn	888	4067	666	666	327
addcontentsto	1066	2301	496	496	267
stack	600	3042	512	514	284
sets	971	3761	622	622	300
cs1	1657	3099	1383	1383	423
pdl2a	1523	2651	1286	1286	417
oopack	166	567	140	140	89
silo	208	531	148	148	106
polyover	131	653	85	85	60
richards	455	934	344	344	155

Table 5: Method Contour Counts

	Analysis			
program	adaptive	1-cfa	local	base
options	12413096	14149096	14169245	14169245
orderedcltn	15566973	29626973	28227122	29627122
addcontentsto	967943	1583828	1583977	1583977
stack	246973	306973	1177122	1127122
sets	23046973	58046973	58047122	58047122
cs1	3204709	3204709	4005523	4005523
pdl2a	6222357	6222537	6377612	6377612
oopack	710523016	1220703514	1220703514	1220703514
silo	11291623	11693779	14338618	14338618
polyover	454847146	1480582168	1480582168	1480582168
richards	11884990	11884990	13334790	13334790

Table 7: Field Read Counts

	Analysis				
program	adaptive	1-cfa	local	base	classes
options	205	150	159	159	90
orderedcltn	125	120	121	121	72
addcontentsto	133	118	118	118	72
stack	123	120	120	120	72
sets	128	119	119	119	72
cs1	222	216	221	221	94
pdl2a	218	215	223	223	94
oopack	33	30	30	30	20
silo	17	16	16	16	21
polyover	13	12	12	12	17
richards	24	16	16	16	24

Table 6: Object Contour Counts

	Analysis			
program	adaptive	1-cfa	local	base
options	204506	278507	284700	284700
orderedcltn	120433	220433	210627	210626
addcontentsto	10444	10468	10661	10661
stack	30432	50432	80625	70625
sets	140432	320432	320625	320625
cs1	645	645	652	652
pdl2a	340633	340643	500634	500634
oopack	13	2018	2018	2018
silo	465668	465670	787899	787899
polyover	41616	149164	149164	149164
richards	141	141	282	282

Table 8: Object Allocation Counts

	Analysis				
program	adaptive	1-cfa	local	base	c++
options	1.97	1.99	1.99	1.99	1.07
orderedcltn	1.79	2.00	1.93	1.93	1.57
addcontentsto	0.05	0.05	0.05	0.05	0.05
stack	0.07	0.08	0.09	0.08	0.09
sets	1.97	1.83	1.88	1.84	1.09
cs1	0.09	0.09	0.08	0.08	0.11
pdl2a	0.68	0.68	0.70	0.70	1.69
oopack	14.9	15.23	18.16	18.16	19.83
silo	1.0	1.0	1.1	1.1	1.8
polyover	12.71	24.43	24.43	24.43	12.83
richards	38	38	40	40	30

Table 9: Runtimes in Seconds

	Analysis				
program	adaptive	1-cfa	local	base	c++
options	628	600	597	597	899
orderedcltn	485	467	466	466	697
addcontentsto	457	450	452	452	686
stack	438	437	437	437	692
sets	483	477	480	480	694
cs1	507	507	503	503	668
pdl2a	485	485	484	484	730
oopack	311	311	310,	310	309
silo	333	333	333	333	342
polyover	316	316	316	316	319
richards	311	311	311	311	313

Table 10: Code Sizes in kB

	Analysis			
program	adaptive	1-cfa	local	base
options	4676	5236	5318	5318
orderedcltn	6538	7818	7700	7700
addcontentsto	164	164	167	167
stack	818	978	1420	1220
sets	15178	17738	17740	17740
cs1	15	15	16	16
pdl2a	12975	12975	14255	14255
oopack	116	132	132	132
silo	12606	12606	15184	15184
polyover	2053	2914	2094	2094
richards	4	4	5	5992

Table 11: Memory Usage in kB