

Automatic Inline Allocation of Objects

Julian Dolby

dolby@cs.uiuc.edu

Concurrent Systems Architecture Group

Department of Computer Science

University of Illinois

1304 West Springfield Avenue

Urbana, IL 61801

Abstract

Object-oriented languages like Java and Smalltalk provide a uniform object model that simplifies programming by providing a consistent, abstract model of object behavior. But direct implementations introduce overhead, removal of which requires aggressive implementation techniques (e.g. type inference, function specialization); in this paper, we introduce *object inlining*, an optimization that automatically inline allocates objects within containers (as is done by hand in C++) within a uniform model. We present our technique, which includes novel program analyses that track how inlinable objects are used throughout the program. We evaluated object inlining on several object-oriented benchmarks. It produces performance up to three times as fast as a dynamic model without inlining and roughly equal to that of manually-inlined codes.

1 Introduction

Traditional object-oriented languages (e.g. SmallTalk [13]) sport a simple, uniform, abstract programming model; all objects are accessed via references and all calls are dynamic message sends. This simplifies programming, as the programmer faces a single model of object behavior, and because different portions of programs can be isolated behind opaque interfaces. This abstract model, however, introduces overhead: even simple field accesses become dynamic dispatches, indirection through references is required to access every object, and individual functions shrink [5]. Hybrid languages like C++ [11] provide multitudinous low-level features (e.g. virtual vs non-virtual functions, inline directives, and explicit stack, inline or heap allocation) so programmers can manually reduce overhead by removing unused abstraction.

However, the original notion of a clean, simple semantics is re-emerging in recent object-oriented languages (e.g. Java [27], NewtonScript [3], Cecil [7]). This leaves eliding abstraction overhead to aggressive implementations. Dynamic dispatches are optimized statically by type inference

[1, 6, 21, 23], dynamically by inline caching [16] or with hybrid approaches like type feedback [17]. Static or hybrid type analysis is combined with method specialization [10, 24] to allow inlining, removing the small functions common in object-oriented code.

The combination of static type analysis and specialization also permits inline allocation of objects within containers, thereby eliminating many object references and reducing object allocation. We present *object inlining*, the first fully automatic optimization for doing inline allocation in an object-oriented language. We exploit the power of analysis and cloning that can handle data flow through object state [23, 24] to inline allocate child objects even for polymorphic containers and to systematically exploit such allocation wherever the child objects are used.

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

class Rectangle {
    lower_left, upper_right;
};
```

Figure 1: A Rectangle class

To make the problem concrete, consider the example in Figure 1¹. A direct implementation is shown in Figure 2(a); inlined allocation, as in C++, would produce Figure 2(b). Accesses to coordinates of rectangles are cheaper in the inlined implementation, requiring one dereference fewer. Furthermore, cache performance is likely to benefit. Inline allocation also reduces object allocation costs, since the sub-objects are allocated with the container. This especially benefits languages like Java where objects have conceptually infinite extent, necessitating heap allocation in general. Note that methods such as `Point::area` in Figure 4 could cache fields with inline allocation of `Points` since `p` and `this` could not be aliased when the method is called from `Rectangle::area`.

Inline allocation creates a correspondence between container and containee that our analysis permits to be exploited as alias information. An example use of this is

¹The syntax used in this paper is reminiscent of C++, but we leave out explicit types to better illustrate our dynamic model

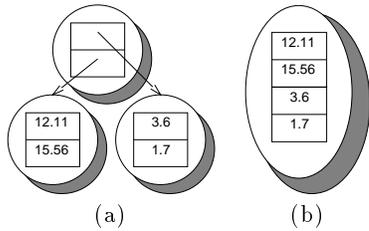


Figure 2: Two Rectangle Representations

caching object fields in registers: more precise aliasing information concomitantly enables more thoroughgoing register allocation of object state as noted above. Furthermore, treating inline allocation as a global program transformation, as we do, allows customized data layouts, such as parallel arrays for an array of objects. We evaluated object inlining on several object-oriented benchmarks, and found it making programs up to three times as fast as without inline allocation, and matching code with inline allocation done by hand.

In subsequent sections, we first introduce our running example (Section 2) and provide background (Section 3). We next cover our analysis (Section 4) and program transformation (Section 5). We then evaluate our optimization (Section 6), discuss related work (Section 7) and conclude (Section 8). We consider future directions in Section 9.

2 Example Program

Our examples of inline object allocation focus on `Points` and `Rectangles` (see Figure 1); the code in this section will be used in our examples hereafter. To illustrate the complexities involved in automatic inline allocation, we introduce a subclass of `Rectangle` (Figure 3). We will use the following methods while expounding various aspects of our optimization; significant methods are in Figure 4, and the main program is in Figure 5.

```
class Parallelogram : Rectangle {
  upper_left;
};
```

Figure 3: Rectangle Subclass

Figure 4 suggests complications of automatic inline allocation that require global program transformations: when the `Point::area` method is called from `Rectangle::area`, it must be specialized because, instead of two point objects, it will be called with one rectangle with four inlined fields. Producing this information requires inter-procedural data-flow analysis (Sections 3.2.1 and 4), and exploiting it is a global transformation.

More difficulties arise when inlined objects are put into unrelated containers: we must call appropriate specialized methods when such objects are accessed via the unrelated container. For example, in Figure 5, the compiler must determine that the call to `head` in `do_rectangle` returns a `Point` inlined into a `Rectangle` so that the appropriate specialized version of `abs` can be called. This requires data-flow analysis that can flow properties through fields (Sec-

```
Point::area(p) {
  return abs(x_pos - p.x_pos) *
         abs(y_pos - p.y_pos);
}

Point::abs() {
  return sqrt(x_pos*x_pos +
             y_pos*y_pos);
}

Rectangle::area() {
  return
    lower_left.area(upper_right);
}
```

Figure 4: Methods

tion 3.2.1). Our Concert [8] analysis framework is, to our knowledge, the only one that can do this.

```
do_rectangle(l1, ur) {
  r = new Rectangle(l1, ur);
  cout << r.area();
  l1 = new List(r.lower_left, nil);
  l2 = new List(r.upper_right, nil);
  cout << head(l1).abs();
  cout << head(l2).abs();
}

main() {
  p1 = new Point(1.0,2.0);
  p2 = new Point(3.0,4.0);
  do_rectangle(p1, p2);
  p3 = new Point3D(1.0,2.0,3.0);
  p4 = new Point3D(4.0,5.0,6.0);
  do_rectangle(p3, p4);
}
```

Figure 5: Main Program

The main program (Figure 5) illustrates our other analysis difficulty. Inlining the points involves copying the fields from the constructor arguments into the inlined `Point` fields within the `Rectangle`. Such copying would become problematic were `do_rectangle` to be called with one aliased `Point` as both arguments, as it would change aliasing relationships. In order to do inline allocation, we must ensure that this has not happened, that is, aliasing relations are not changed so inlining is safe.

In sum, our example illustrates the two challenges we face: 1) finding and specializing uses of inlined objects and 2) ensuring that inline allocation does not change aliasing relationships.

3 Background

In this section, we discuss manual inline allocation in the context of C++. Then we present the compiler framework for our optimizations, focusing upon the sensitivity that it permits: its ability to handle data flow through object fields is vital to our analysis.

3.1 Manual Inline Allocation

C++ provides a direct syntactic mechanism for specifying whether a given object slot should be inlined: the type system explicitly denotes storage allocation. However, the type system denotes other things too, so specifying an object as inline allocated (e.g. declaring `lower_left` to be of type `Point` rather than `Point *`) also determines other behaviors (e.g. changes the meaning of assignment). Hence inline allocation is not simply a performance optimization. This can make specifying inline allocation awkward—for instance a list element conceptually contains a *reference* to its data—or even impossible due to semantic changes. Finally, this mechanism leaves the burden of deciding what to inline on the programmer. On the other hand, because we want to preserve a uniform model and our analysis opens up other optimization opportunities, our inline allocation is done automatically by the compiler.

3.2 The Concert Compiler

This work was done in the Concert [8] compiler, so subsequent discussion of our optimization relies heavily on the framework of that compiler. Therefore, a brief overview is given of the two portions that we use: the analysis and cloning modules.

3.2.1 The Analysis Framework

The Concert compiler has a global analysis framework (see [23, 22] for more detail) that does context sensitive flow analysis. Context sensitivity adapts, in a demand-driven manner, to program structure. A *method contour* [26] represents execution environments for a method, i.e. it is the unit of context sensitivity. Method contours can discriminate arbitrary data-flow properties of its *caller* and *creator*. An *object contour* represents method contours of the statements that created a given object. For method contours:

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 creation point’s method contours.

Contours are created on demand: they are created when the analysis needs to distinguish some property. 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.

Figure 6 illustrates type analysis on part of the program in Figures 4 and 5. In Figures 6 and 7, contours are signified by “[<callers>, <creators>].” The first graph is after one pass of analysis; the calls to `do_rectangle` have different argument types, so two contours are created to distinguish them; on the other hand, since both calls to `Rectangle::area` have the same types for their arguments, just one contour is created embracing both call sites. Within `Rectangle::area` the type

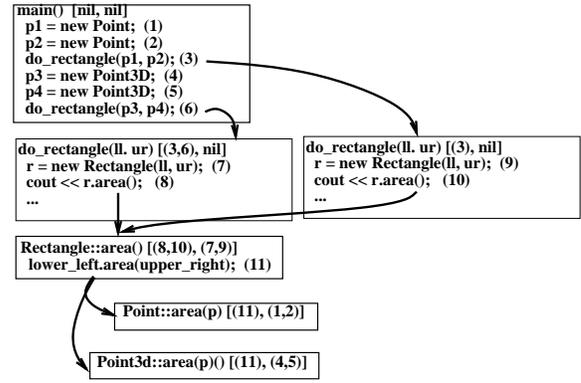


Figure 6: Pass One

of `Rectangle::lower_left` is ambiguous, so the demand-driven specialization comes into play. Since the type confusion is due to a field, the system creates object contours representing each creation statement of `Rectangle` to distinguish the types of `Rectangle::lower_left`. This results in Figure 7.

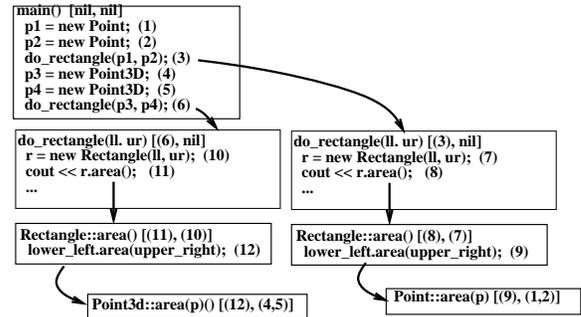


Figure 7: Pass Two

3.2.2 The Cloning Framework

The Concert compiler includes a mechanism for cloning based upon data-flow properties discovered by analysis [24], which works on the contours. The mechanism uses a generic function that determines, for two given contours, whether they are compatible. Cloning is done upon both methods and classes.

For each method in the program, its method contours are grouped into sets of compatible contours, and a copy of the method is generated for each set. Caller information from the contours is used to reconstruct the call graph. The original use of this mechanism is to eliminate dynamic dispatches from the program, and so it marks contours as incompatible if they have different types for the target of any call in the method.

For example, the method `Rectangle::area` in Figure 7 has two contours to discriminate the type of the `lower_left` field of `Rectangle`. In the program, however, there is only one method, requiring that `lower_left.area(upper_right)` be dynamically dispatched. The cloning mechanism marks

these two contours as incompatible as their types for the field `lower_left` of `Rectangle` differ; then, `Rectangle::area()` is duplicated and `lower_left.area(upper_right)` can be statically dispatched in each one since the type of `lower_left` is precise.

Note that cloning `Rectangle::area` creates problems in `do_rectangle`: there are now two possible methods to call for `Rectangle::area`. The cloning framework includes an iterative mechanism to split caller methods when cloning a callee creates a dynamic dispatch, which in this case splits `do_rectangle` as well.²

Cloning is also done on classes: classes are split based upon the object contours. As with methods, cloning works by grouping the contours based upon some discriminator function. Cloning a class does not necessarily require cloning all associated methods.

4 Object Inlining Analysis

As Section 2 illustrated, automatic inline allocation requires two analyses: first, all uses of fields in inlined objects must be found precisely so that the appropriate inlined field can be used instead; we call this use specialization. Second, assignments to the reference field being removed must be found, and we must verify that converting these to assignments to update the inlined fields is safe; we call this assignment specialization.

4.1 Use Specialization

To determine from which object fields any given value in the program may have come, we tag values that result from field accesses and propagate these tags through the inter-procedural data-flow graph. The basic idea is that values are tagged with the names of any fields from which they may have come, and these tags are transitive on field accesses to objects that themselves were the result of a field access. More formally, our data-flow analysis framework gives us the following properties of each value in the program:

Backs(v) are the values from which data flows into v

Creators(v) is the set of object contours of v .

We also define tags with which to mark the results of accesses to fields; these tags keep track of the fields from which a given value might have come. Fields themselves are represented by special values that denote the contents of the field. Note that `MakeTag` can build nested tags for accesses to nested objects.

NoField is a special tag for values that did not flow from field accesses

MakeTag(f, tag) returns the tag representing values that came from field f from an object whose origin is represented by tag .

Head(tag) returns the last field in the given tag, i.e. $Head(MakeTag(f, tag)) = f$.

Tags(v) is the set of tags associated with the value v .

²The situation gets even worse here because the dispatch of `Rectangle::area` depends on the type of a slot, which would make the call graph unrealisable if `do_rectangle` were not split.

We have three different transfer functions for our data-flow analysis, corresponding to three cases: object creations, instance variable accesses and other operations. Object creations are places where new objects are allocated (i.e. `new` statements) and their result values get the special `NoField` tag.

$$v = \text{new Object} \implies \text{Tags}(v) \leftarrow \{\text{NoField}\}$$

Instance variable accesses append the accessed field onto the existing tags of the object being accessed; thus v gets a tag representing all the fields accessed to obtain this value.

$$v = o.f \implies \text{Tags}(v) \leftarrow \bigcup_{t \in \text{Tags}(o)} \text{MakeTag}(f, t)$$

where f is the special value for the given field. Other values get their tags by propagation from all their reaching definitions in the inter-procedural graph; the use of the `Creators` of the tag prevents extraneous propagation that would otherwise occur across dynamic dispatches.

$$\text{Tags}(v) \leftarrow \bigcup_{i \in \text{Backs}(v)} \left\{ t \mid t \in \text{Tags}(i) \wedge \text{Creators}(\text{Head}(t)) \cap \text{Creators}(v) \right\}$$

Marks from different slots, or from no slot, can converge whenever data-flow paths do, and the analysis uses the contours provided by the framework to split such paths. As method contours are created on demand, the analysis must assert when two calls to the same method require different contours; this happens when two objects with different tags are supplied at different call sites to the same argument. That is, c_1 can be in the contour of c_2 only if

$$\forall i \text{Tags}(\text{Arg}(c_1, i)) \subseteq \text{Tags}(\text{Arg}(c_2, i))$$

where $\text{Arg}(c, i)$ represents the i th argument of call c . An example situation that requires splitting is the two calls to `abs` in `do_rectangle`; it is illustrated in Figure 8.

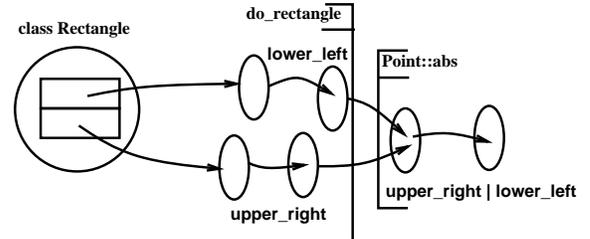


Figure 8: A Call Confluence

We also use creator sensitivity to disambiguate values with different tags that flow into an object contour's state. This arises in the two `List` objects created in `do_rectangle`, and is illustrated in Figure 9. The object contour must be split into new ones, one for each distinct tag amongst the definitions. Thus, the definitions of contour o must be partitioned into contours o_n such that

$$\forall f \in \text{Fields}(o) \forall v_i, v_j, v_i, v_j \in \text{Backs}(o.f) \\ (\exists o_n v_i, v_j \in \text{Backs}(o_n.f)) \implies \text{Tags}(v_i) = \text{Tags}(v_j)$$

Once the definitions have been partitioned thus, the analysis framework will split the object contour if possible.

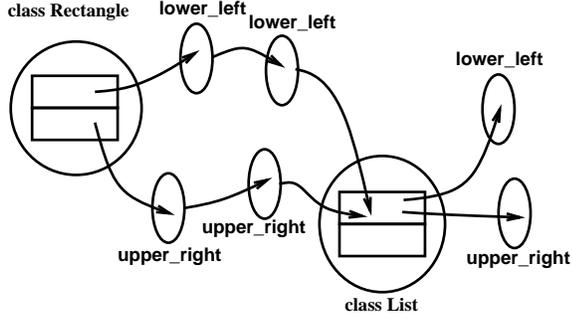


Figure 9: Field Confluences

This analysis marks each value with an approximation of from which fields it may have come. We use this information to specialize the program to work on inline-allocated objects: a field can be inline allocated only if this analysis is able to distinguish exactly where the given field is used. That is, the tags of the given field must not be confused with tags from any other field.

4.2 Assignment Specialization

Copying the contents of an inlined object into its container potentially alters aliasing relations, and we must ensure that such copying is safe. A common case is an object is initialized and then assigned to an inlined slot and from then on is accessed via the container (this happens in Figure 5). To handle such cases, we analyze to determine method arguments that can be passed by value; if the argument to the field's mutator³ method may be passed in by value, we can copy it into the inlined fields safely.

Our analysis is defined in terms of contours, which represent method calls, edges which map arguments between caller and callee, and uses, which are primitive operations, in particular calls and assignments. The basic idea behind this analysis is that objects may be passed by value if they have not previously been stored and are not subsequently used. We first define some local operations upon specific values and contours that form the groundwork for our analysis:

Caller(e) is the calling use of inter-procedural edge e .

Callee(e) is the called contour of inter-procedural edge e .

Call(u, v) yields true if the use u of value v is a call.

Edge(u) is the inter-procedural edge corresponding to use u , assuming u is a call.

Uses(v) is the set of uses of a value v in its contour; these uses represent calls and primitive actions like setting a variable.

UsesBefore(e, v) is the set of possible uses of v in $Caller(e)$ before the edge e .

UsesAfter(e, v) is the set of possible uses of v in $Caller(e)$ after the edge e .

Map(v, e) gives the argument value in the called contour corresponding to the value v passed across edge e .

LocalCreation(v) returns true if v is the product of a new operation within its contour.

DontStore(u, v) is true if the use u does not store v in an instance variable or a global variable.

Given these operations, we define a *NoStore* predicate that is true if a given value v is not stored in persistent state (all such state in our model is either an instance variable or a global variable).

$$\begin{aligned}
 NoStoreCall(u, v) &\leftarrow \\
 &Call(u) \wedge \\
 &NoStore(Callee(Edge(u)), Map(Edge(u), v)) \\
 NoStoreUse(u, v) &\leftarrow \neg Call(u) \wedge DontStore(u, v) \\
 NoStore(u, v) &\leftarrow \\
 &NoStoreCall(u, v) \vee NoStoreUse(u, v) \\
 NoStore(c, v) &\leftarrow \forall p_i \in Uses(v) NoStore(p_i, v)
 \end{aligned}$$

Valuability for locally created objects (results of **new** operations within a given contour) can be checked by looking at all the uses of such objects within the contour. Once an object is passed in by value, we can copy it, and hence it is, effectively, created locally. The following helper predicate reflects this:

$$\begin{aligned}
 CreatedLocally(v) &\leftarrow \\
 &LocalCreation(v) \vee CallByValue(v)
 \end{aligned}$$

We are now ready to define valuability: an argument is callable by value if it can be passed by value from each call site. *PassByValue* and *CallByValue* formalize the criterion set out above:

$$\begin{aligned}
 PassByValue(p, v) &\leftarrow \\
 &\forall p_i \in UsesBefore(p, v) \left(\begin{array}{l} NoStore(p_i, v) \wedge \\ \neg UsesAfter(p, v) \wedge \\ CreatedLocally(v) \end{array} \right) \\
 CallByValue(v) &\leftarrow \\
 &\forall p_i, v_i \{ (p_i, v_i) \mid Map(Edge(p_i), v_i) = v \} \\
 &PassByValue(p_i, v_i)
 \end{aligned}$$

A given field may be inlined only if the mutator method has its value argument passed in by value.

5 Object Inlining Transformation

Once analysis has determined which objects are inlinable, we transform the program. In discussing our transformation, the *inlined field* is the field being removed and the *inlined state* is the fields being added to the container. First, the methods and classes needed for specialization must be created by cloning. Then classes must be restructured to remove inlined fields and add new fields for inlined state; all uses of inlined fields must be redirected to the container's new inlined state. Finally, definitions of inlined fields must be altered to update inlined state instead.

³all access to field go thru accessor functions in our model

5.1 Cloning

Analysis has marked all values indicating from which fields they came. Thus, to specialize methods on inlined fields, we use the Concert cloning mechanism (Section 3.2.2) to separate method contours that come from differing fields. Thus, we get one clone of each method for each inlined field on which it could be called. This requires just checking whether, for two given contours, the field from which “self” could have originated is the same. The cloning is illustrated in Figure 10

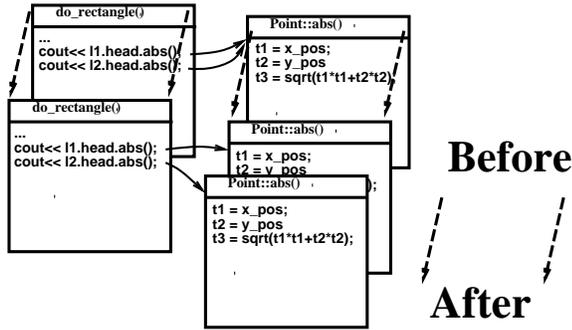


Figure 10: Object Inlining Code Cloning

Similarly, we clone classes to handle polymorphic containers. Suppose that a polymorphic field is inlinable according to our analysis; the type analysis (See Section 3.2.1) creates object contours for each type of the polymorphic field (See Figure 7). To inline fields of multiple types, we must have distinct container classes, and so cloning splits object contours corresponding to different types of inlinable fields into different classes.

5.2 Restructuring Classes

When adding inlined fields to a class, we must avoid altering the layout of other fields of the class. Consider a polymorphic container in which an inlined field has different types in different situations: it has been cloned, but we need not clone methods that do not use the inlined field if we can preserve the layout of the other fields across all the cloned classes. Furthermore, the layouts of the container class and any subclasses must remain conforming. Both constraints can be satisfied by replacing the inlined field with one field from the inlined class, and adding the rest of the fields at the end of the fields of the container class. This is illustrated in Figure 11

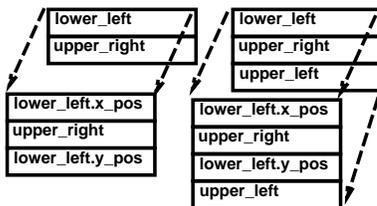


Figure 11: New Rectangle and Parallelogram

5.3 Use Specialization

Field references whose object is from an inlined field must be redirected to the corresponding inlined state. This involves two steps: traversing all the methods called upon the inlined field, adjusting field references to use the new inlined state, and eliding accesses to the inlined field in methods on the containing object. The effect of this transform is illustrated in Figure 12.

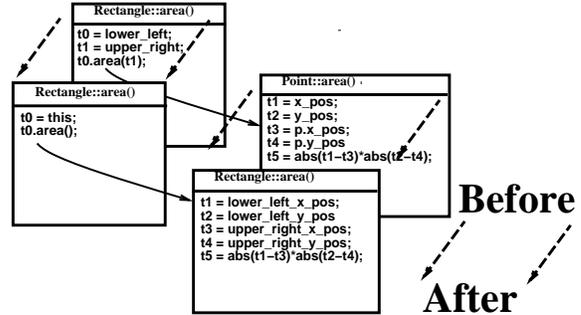


Figure 12: Code Changes for Rectangle and Point

Inlining arrays (converting arrays of references to arrays of object states) presents a complication. Uses of the inlined field become array accesses to the inlined state, and so we must remember what array index to use. When eliding accesses to inlined array contents, we pass the array index used along with the containing array to uses of the inlined field. This index value is then used when altering the accesses to the inlined field by turning them into array references. This is illustrated in Figure 13.

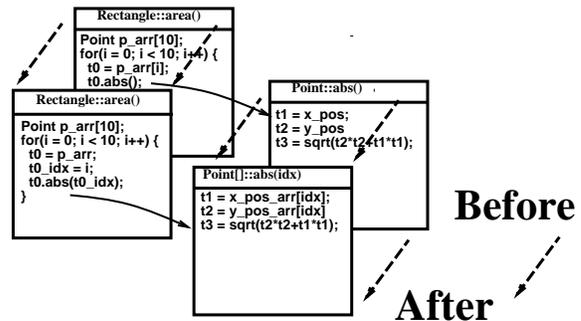


Figure 13: Code Changes for an Array of Points

5.4 Assignment Specialization

When the inlined field is removed, assignments to it must be converted into assignments to the inlined state. Analysis ensures this is safe, so we replace the original assignment to the inlined field with copies into the inlined fields; we traverse methods on the containing object, changing each assignment. Array contents are special: copies are generated for each array element (each array has a field that records its size).

6 Evaluation

To evaluate our optimization, we implemented it in the Concert compiler, and ran it with a suite of object-oriented benchmarks. We report on the effectiveness, costs and benefits of our optimization. Effectiveness covers whether our analysis can inline all slots for which it is appropriate; the benefits are performance gains. The costs are both compile time and code size; we quantify compile time by measuring the additional analysis sensitivity object inlining requires. We choose the following benchmark programs to stress all aspects of our implementation:

OOPACK is a set of kernels from KAI that demonstrate the compiler’s ability to perform simple object-oriented optimizations. One kernel (the **ComplexBenchmark**) uses arrays of complex number objects; these numbers are inline allocated in C++, but would be references in Java or Lisp. Our transformation inlines these objects into their containing arrays. We include timings only for this kernel.

Richards is an operating system simulator benchmark. It has some array objects that can be inlined into containing objects; more interestingly, the Task object has a private data pointer (declared as `void *` in C++ and accessed using casts). Various subclasses use different types in this slot, and hence it cannot be declared inlined in C++. Our transformation inlines the private data independently for each subclass.

Silo is an event-driven simulator benchmark from the repository at Colorado⁴. Some wrapper objects for queues can be inlined into their containers, and list items (essentially `cons` cells) can be eliminated by combining them with their data. The queue wrappers are inline allocated in C++, but the `cons` cells cannot be.

polyOver is the benchmark from [28]; it computes an overlay of two polygon maps; it uses several algorithms employing arrays and lists of polygons. Our transformation inlines `cons` cells as in Silo, contents of arrays, and, most interestingly, an array of `cons` cells. These cells stored references to each other, and hence inlining them requires our analysis’ ability to flow thru data. The arrays are inline allocated in C++, but the `cons` cells cannot be.

These benchmarks are all pre-existing C++ codes; since the Concert Compiler accepts ICC++ [9], a C++-derived language, only minor changes were required for our compiler to accept them. These were mostly declaration and array access syntax. However, the Concert Compiler assumes an object model in which all objects are accessed via references, so syntactic inline allocation is ignored.

6.1 Analysis Effectiveness

Given our approach of not (conceptually) destroying the containee, the only fundamental limit on what objects our optimization can inline is the ability to add copies. Adding copies depends upon aliasing information, so is necessarily conservative. For each of the codes, we present four field counts: the total number of fields which hold objects,

the number that could ideally be inlined given aliasing constraints (determined by hand), the number of fields declared inline in C++, and the number fields our optimization inlined.

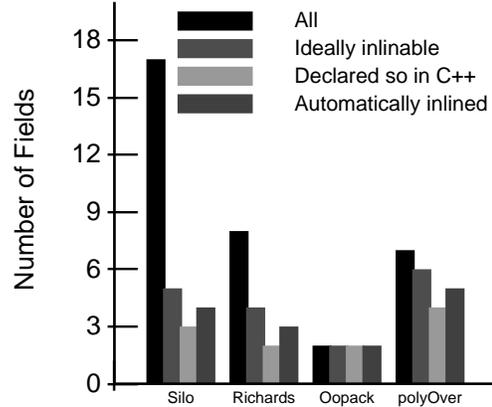


Figure 14: Inlinable Field Counts

Our analysis did as well or better than manual inline allocation on all codes; there was no field manually declared inline in C++ that our analysis did not find inlinable. We did better than C++ on Silo, Richards and polyOver. This is because we inlined a polymorphic field in Richards and merged `cons` cells with their data in Silo and polyOver.

In both polyOver and Silo, lists present problems. In Silo, our analysis cannot inline `cons` cells of the global event list, because it cannot tell that a given event is in the list at most once; hence it thinks there is possible aliasing between the data of different elements, so the data and list element cannot be merged. In polyOver, a list cannot be blocked because it is constructed in a loop, and our analysis has no mechanism to distinguish loop iterations.

The Richards code causes another difficulty: an array of pointers to tasks. The array is polymorphic (different elements reference different types of task) and our analysis does not distinguish different array elements, simply representing the array state as an instance variable.

6.2 Optimization Cost

There are two obvious costs of object inlining: the extra sensitivity required by the analysis and the accretion of extra generated code from cloning. While the analysis does require a good deal of extra sensitivity, and produces many more clones than without it, this does not translate into more generated code.

6.2.1 Generated Code Size

The Concert Compiler generates C++ code as a portable assembly language; for this reason, among others, our code is much larger than that of G++, but object inlining does not increase the size of the code, but rather the opposite, as Figure 15 shows. This table measures the size in kilobytes of stripped object files generated by G++.

The reason is twofold: by removing object allocations and heap references, object inlining shrinks the size of spe-

⁴<http://esl.cs.colorado.edu/C++BenchmarkSuite/root.html>

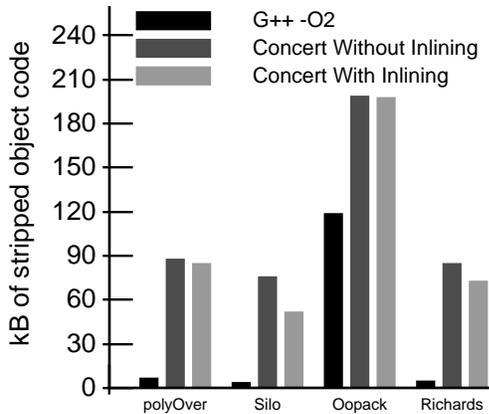


Figure 15: Object Inlining Code Expansion

cialized methods compared to the original program. Furthermore, most of the specialized methods are inlined, so the cloned methods are not generated by themselves anyway.

6.2.2 Analysis Cost

We present analysis sensitivity required as a metric of our analysis' cost. As our analysis works by creating different contours for uses of object fields, a good measure of its cost is a count of the number of contours created with and without object inlining. Our compiler performs other analyses, so more contours than methods are needed even without object inlining. Figure 16, presents the number of method contours required per method in the program as a measure of precision required.

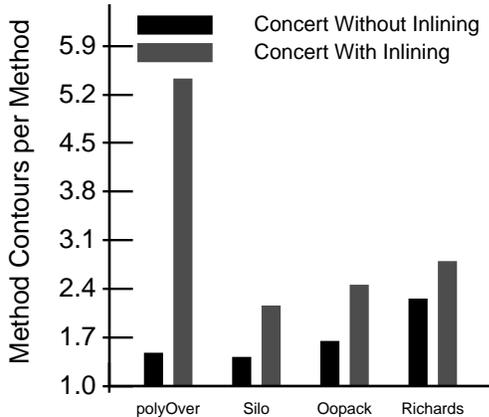


Figure 16: Method Contours Required

Our analysis framework also creates object contours for handling data flow thru state; the object inlining analyses did not, for these programs, require additional object contours. Even polygon overlay does not generate more object contours, despite requiring analysis through object fields, because the fields used contain differing types, so the type

inference mechanism splits the relevant object contours even when object inlining does not. The polygon overlay code also causes the largest increase in required sensitivity, mostly because essentially all the data structures in this program are transformed.

6.3 Performance

To measure the performance improvement brought about by object inlining, we compiled several our chosen object-oriented benchmark programs using our Concert compiler both with and without object inlining; to provide calibration, we also compiled the same programs with G++; G++ was used with -O2 for both the C++ programs and the Concert compiler generated code. Measurements were taken on a SparcStation 20/60, and are the average of 10 runs.

Figure 17 is normalized to the performance of Concert code without object inlining; it shows that the Concert System, without inlining, gives roughly similar performance to G++ except on polyOver. Our use of C++ as a portable assembly language, unambitious array optimization, and, naturally, the cost of accessing uninlined objects are the major contributors when Concert is slower; a highly-tuned memory allocator is the major reason Concert is faster in Silo.

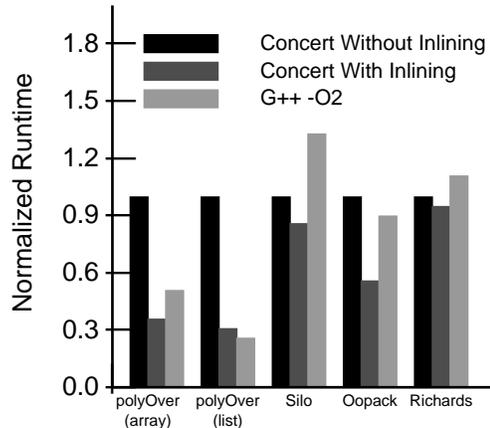


Figure 17: Object Inlining Performance

The performance gain of object-inlining is most dramatic on polyOver: both the array and list versions are roughly three times as fast as without it. This code boasts the most aggressive use of inlining: polygons are inlined into arrays, tightening inner loops; result polygons are merged with the `cons` cells of their list, reducing dynamic allocation; and a list of `cons` cells is inline allocated, which also tightens loops. The combining of resultant polygons and `cons` cells produces tighter data-structures than the C code, which is why the array version is faster than G++; the list version should be faster for the same reason, but low-level code generation issues in the Concert compiler frustrate this.

OOPACK is nearly twice as fast with inlining as without, and it ends up substantially faster than G++. This is due, in part, to inlining laying out the complex number array as parallel arrays (Fortran style) rather than by object, which seems to improve cache performance for this code. The 14% gain for Silo comes primarily from reducing dynamic allocation by merging `cons` cells with their data. Richards creates

very few objects, and the 5% gain it shows derives partly from eliding pointer dereferences. Both Silo and Richards benefit from improved object field caching with object inlining.

Thus, object inlining overall makes code run up to three times as fast as without inline allocation and matches the performance of code with inline allocation specified by hand.

6.4 Discussion

Our optimization manages many cases, from simple situations where a C++ programmer would use inline allocation to more complex situations like eliding `cons` cells, which would, at a minimum, cause conceptual disruption to the code if a C++ programmer used inline allocation. `polyOver` is a low-level C program with carefully tuned storage management resulting in messy combinations of pointers. Our analysis is able to reproduce even such ugliness as that automatically. Furthermore, doing so provides substantial performance gains without bloating code size, indeed the code usually shrinks a little bit.

The major limitation is, as one might expect, aliasing information needed to make inlining decisions. The limitations in Richards and `polyOver` are implementation issues, better loop and array analysis respectively could solve these without fundamental difficulty. On the other hand, Silo is more problematic: to eliminate the `cons` cells of the event list would require strong aliasing information. Given the difficulty of alias analysis, this represents a limitation on our optimization's efficacy.

7 Related Work

The idea of rearranging data layout to improve performance is widespread, and the work most relevant to ours comes from three camps: the object-oriented realm, the functional community and Fortran array optimization.

Object-Oriented Systems The impetus for object inlining came from the ability to specify such inlining by hand in hybrid-object languages like C++ and Oberon-2 [14, 20] that distinguish between objects and object references. But the idea of doing inline allocation automatically is by no means new; the Emerald [4, 18] object system was designed with this goal in mind. Indeed, they wanted to automatically generate specialized representations of objects and transform the code that used them. This is exactly what we do. However, their compiler was not capable of the requisite analysis (Section 4). The compiler did do type inference, and they could inline allocate immediate types when they had precise type information. They could do this because immediate types in Emerald, as in most object-oriented languages, are values—their contents are important but their identities are not—so they could avoid our analyses by simply copying in and out of the field.

Functional Languages There has been much work in the functional community on *unboxing*, in which specialized representations are used to reduce storage and access overhead. Of particular relevance to our work is unboxing in the presence of polymorphism [19, 15]. In [15], Cordelia Hall and company present a transformation for Haskell that, upon being told what variables to unbox, generates specialized

code to exploit the unboxing, even for polymorphic functions. They generate specialized function versions using a partial evaluator to propagate “unboxedness” through the program. The propagation of “unboxedness” resembles the propagation of tags our use analysis defines; however, it only works for immediate types.

In [25], Shao et al. unroll linked lists—essentially inline allocating tail pointers—in a functional subset of ML. Their analysis works using *refinement types* [12] 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 analysis is similar in using an abstract interpretation, but our field tags are more general, as they handle arbitrary object structures, rather than lists. Also our scheme, because our abstract interpretation is interprocedural, only analyzes specializations that are actually used, whereas theirs analyzes all possible combinations.

Since this work was done for functional languages, their analysis can be more straightforward than ours: it need not handle structure assignment, i.e. data flow through containers. This provides two simplifications: it eliminates the difficulties created by assigning inlined object into other containers, and it obviates the need to prove inline allocation is safe.

Fortran Optimizing array layout for cache performance [2, 29] also involves transforming data layout. Arrays and loops that operate upon them are *tiled*; that is, the inner loops are strip-mined to operate upon small portions of the arrays that will fit in cache. Because array iterations cannot be reordered arbitrarily—due to data dependence—the arrays are rearranged in memory so these small portions exhibit spatial locality. Altering a given array's layout naturally affects all code that uses it, and finding all such uses is analogous to our use specialization analysis, although much simpler analyses suffice for common Fortran code. This alteration of array layout is typically done only when it can be expressed as an affine transformation on the array indices and the analysis is generally not context sensitive.

8 Conclusions

We presented *object inlining*, an optimization that automatically inline allocates objects, allowing us obtain similar efficiency to a system with explicit inline allocation while preserving the programmability benefits of a uniform object model. To do this requires a novel analysis and transformation mechanism that builds upon previous work on analysis and specialization of object-oriented programs. Our evaluation shows that our analysis can handle not only situations where inline allocation is done in C++, but also situations where such manual allocation would be conceptually disruptive. We showed several object-oriented benchmarks running up three times as fast due to object inlining as opposed to without it, and matching the performance of code with inlining specified by hand.

9 Future Work

Turning our analysis loose on C++ demonstrates that it can, at least, discover inlining opportunities in the sort of places

that C++ programmers specify them, and provides a basis for comparison in standard C++ compilers such as G++. However, the greater opportunity for automatic inline allocation is languages with a more abstract object model that do not allow its explicit specification. These languages also pose the greater challenge by promoting a more polymorphic and dynamic kind of program. But our optimization's handling of polymorphic containers and `cons` cells suggest such dynamic codes will be doable. We intend to apply our optimization to such languages, Java in particular.

Acknowledgments

Others have contributed to this paper and the work it describes. My advisor Professor Andrew Chien provided a great deal of helpful feedback on the structure and content of the paper; the Concert System has been the work of Andrew Chien, Hao-Hua Chu, Bishwaroop Ganguly, Vijay Karamcheti, John Plevyak and Xingbin Zhang as well as myself. In particular, the analysis framework upon which this work is based was largely designed and implemented by John Plevyak.

The research described in this paper was supported in part by DARPA Order #E313 through the US Air Force Rome Laboratory Contract F30602-96-1-0286, NSF grant MIP-92-23732, ONR grants N00014-92-J-1961 and N00014-93-1-1086, NASA grant NAG 1-613, and supercomputing resources at the Jet Propulsion Laboratory. Support from Intel Corporation, Tandem Computers, Hewlett-Packard, and Motorola 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] Jennifer M. Anderson, Saman P. Amarasinghe, and Monica S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of Fifth Symposium on Principles and Practice of Parallel Programming*, 1995.
- [3] Apple Computer Inc. *The NewtonScript Programming Language*, December 1995. Available online from ftp://ftpdev.info.apple.com/Developer_Services/Newton_Development/DOCS_PDF/NSCRIPTR.ZIP.
- [4] 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.
- [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] Craig Chambers. The Cecil language: Specification and rationale, version 2.0. Technical report, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, March 1995.
- [8] 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*, April 1997.
- [9] Andrew A. Chien, Uday S. Reddy, John Plevyak, and Julian Dolby. ICC++ – a C++ dialect for high-performance parallel computation. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*, March 1996.
- [10] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 93–102, La Jolla, CA, June 1995.
- [11] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.
- [12] 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.
- [13] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1985.
- [14] H. Mossenbock. *Object-Oriented Programming in Oberon-2*. Springer-Verlag, 1993.
- [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] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th Symposium on the Principles of Programming Languages*, pages 177–188, 1992.
- [20] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison Wesley, 1992.
- [21] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA '91*, pages 146–61, 1991.

- [22] John Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1996.
- [23] 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.
- [24] 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.
- [25] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *ACM Conference on Lisp and Functional Programming*, June 1994.
- [26] Olin Shivers. *Topics in Advanced Language Implementation*, chapter Data-Flow Analysis and Type Recovery in Scheme, pages 47–88. MIT Press, Cambridge, MA, 1991.
- [27] Sun Microsystems Computer Corporation. *The Java Language Specification*, March 1995. Available at <http://java.sun.com/1.0alpha2/doc/java-whitepaper.ps>.
- [28] Gregory V. Wilson and Paul Lu, editors. *Parallel Programming Using C++*. MIT Press, 1995.
- [29] Micheal E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1991.