An Automatic Object Inlining Optimization and its Evaluation

Julian Dolby*
dolby@us.ibm.com
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598-0704
+1 (914) 784-7299

Andrew A. Chien
achien@cs.ucsd.edu
Department of Computer Science and Engineering
University of California, San Diego
+1 (858) 822-2458

Abstract

Automatic object inlining [19, 20] transforms heap data structures by fusing parent and child objects together. It can improve runtime by reducing object allocation and pointer dereference costs. We report continuing work studying object inlining optimizations. In particular, we present a new semantic derivation of the correctness conditions for object inlining, and program analysis which extends our previous work. And we present an object inlining transformation, focusing on a new algorithm which optimizes class field layout to minimize code expansion. Finally, we detail a fuller evaluation on eleven programs and libraries (including Xpdf, the 25,000 line Portable Document Format (PDF) file browser) that utilizes hardware measures of impact on the memory system.

We show that our analysis scales effectively to large programs, finding many inlinable fields (45 in xpdf) at acceptable cost, and we show that, on some programs, it finds nearly all fields for which object inlining is correct, and averages 40% of such fields across our benchmarks. We implement our analyses in an advanced analysis infrastructure, and we show that, compared to traditional 1-CFA, that infrastructure provides better results and lower and more scalable cost. Across all programs, analysis identified about 30% of objects as inlinable on average. Our transformation increases code size by only 20% while inlining this 30% of fields. Inlining these objects eliminated on average 28% of field reads, 58% of object creations, 12% of all loads. Further, the optimized programs have significantly improved memory reference behavior, producing 25% fewer L1 data cache misses and 25% fewer read stalls. On average the runtime improved by 14%.

1 Introduction

Object-oriented languages and frameworks aid the construction of large software systems by enabling extensive software reuse, generic programming, and composition of components. These techniques use abstraction—layers of objects and interface methods—to allow clean composition of modules. As a result, such techniques produce programs with deeply layered interfaces, unspecialized data structures, and large numbers of small objects and methods. Such techniques are well represented in contemporary object oriented languages [11, 43, 40, 41], component software frameworks [27] and libraries [24, 39].

Providing isolation between software modules requires an interface semantics that hides implementation details. Direct implementations of such a semantics can be disastrous for performance. For example, in an object-oriented language such as Java, simply retrieving a value can go from a load instruction to a dynamically-dispatched message send on a heap object. Interface overheads are often exhibited as slower and more frequent procedure calls and pointer dereferences. The overheads induced by indirect method calls are well-studied [3, 1, 9, 31, 35, 28, 29, 16, 36], and both static and dynamic optimizations developed by our group and others in the community can dramatically reduce such procedure calling overhead.

But modular software design also incurs inefficiencies in data structure design and layout, because it encourages composing generic structures rather than building custom optimized ones. General data structure libraries such as the Java standard library and NIHCL [24] are examples of this phenomenon. Such overheads have been addressed before: allocations by sophisticated memory management [4, 21] and by storage analyses [7, 42], and dereferences by prefetching [33] and by redundant load and store elimination [18], which uses sophisticated alias analyses [23, 17].

We are exploring the problem of increased object indirection by developing automatic object inlining optimizations. These techniques preserve the programming model's modularity and semantics while automatically transforming the programs' data structures into a more efficient implementation. Object inlining fuses objects together when they have a parent-child relationship, reducing object allocation, access and dereference cost, and generally improving program memory behavior. The contributions of this paper build on our previous work [20, 19], but provide a significant advance in capability. Specifically, this paper incorporates the following novel contributions:

1. A formal description object inlining analysis, with necessary and sufficient conditions for semantics preservation. This description exposed several significant limitations in our previous formulation, enabling their correction.

[*] Work done as a Ph.D. student at the University of Illinois at Urbana-Champaign

PLDI 2000, Vancouver, British Columbia, Canada.
Copyright 2000 ACM 1-58113-199-2/00/0006...$5.00.

2. A new algorithm for the object layout problem which arises when objects are inlined. We evaluate this algorithm empirically, and show that it is effective in maintaining layout conformance, limiting the required method cloning to only 20%, despite inlining 30% of all objects.

3. A detailed empirical evaluation including large programs (up to 25,000 lines) which shows surprisingly large numbers of fields to be inlinable (12 - 40% of all object fields) and consistent runtime performance improvements ranging from 4 - 29% with a mean of 14%. We also characterize compiler optimization cost and effectiveness. We believe these are the largest object-oriented programs which have ever been subjected to this type of aggressive whole-program analysis.

4. A detailed understanding of how these high level optimizations affect performance thru memory performance statistics obtained from hardware performance counters. These statistics show clearly that the key factor is improved program memory behavior which produces a 25% reduction in cache misses and 25% reduction in both read and write stalls.

We start in Section 2 with examples illustrating object inlining and motivate the program information required to perform object inlining. Next, we present the semantic underpinning of our object inlining optimization in Section 4; the analysis derived from this base is defined in Section 5. We discuss issues of the transformations in Section 6. We show empirical results in Section 7, summarize related work in Section 8 and finally conclude in Section 9.

2 A Program Example

To illustrate the effect of object inlining, Figures 1 and 2 show an inventory program[1] before and after object inlining.[2] The original program (Figure 1) contains two classes, an `Item` class (which records an item's product id, department, and value) and an `ItemList` class (a generic linked-list with a `total` method which computes inventories' total value). The inventory itself is global variable, `inventory`, of a list of `Items`. The inlined program (Figure 2) contains just one class, `ItemListAndItem`, with all methods.

To be sure the object inlining transformation is semantics preserving, we must be sure that each `datum` associates each `ItemList` with exactly one `Item`. In addition, since we are potentially modifying the data layout, we must be able to locate all program points where the `ItemList` and `Item` state is used. With safety ensured, the inlining transformation consists of three steps:

1. build class with state from `Item` and `ItemList`,

2. delete creations of `Item` and `ItemList` and replace with creation of the fused `ItemListAndItem`, and

3. modify uses of `Item` state to use fused object state.

In the following sections, we discuss how to employ this technique rigorously and in a general setting.

```
class Item {
  int product_id;
  int department;
  float value;

  Item(int p, int d, float v) {
    product_id = p;
    department = d;
    value = v;}

  double value() { return value; }
};

class ItemList {
  Item *datum;
  ItemList *next;

  ItemList(ItemList *d, ItemList *n)
    datum = d;
    next = n;  }

  double value() {
    if (!next)
     return datum->value();
    else
     return
      datum->value()+next->value();
  }
};

ItemList *inventory = NULL;

void AddItem(int id, int d, float v) {
  Item *item = new Item;
  item->Item(id, d, v);
  ItemList *l = new ItemList;
  l->ItemList(item, inventory);
  inventory = l; }
```

Figure 1: Inventory Program (before Object Inlining)

3 Concert Compiler

Our object inlining analysis and optimization implementation is done using the Illinois Concert System [10], a vehicle for research into advanced implementations for object-oriented languages. It implements the language ICC++ [11], a language with C++-like syntax and Java-like semantics, which we use for our benchmarks. It also provides an aggressive analysis framework [35, 34], with two novel features: *iterative adaptive* analysis repeatedly analyzes a program, incrementally adding context sensitivity as needed for information precision; this allows aggressive analysis to be performed efficiently. In Section 7.5, we will evaluate object inlining implemented in both this advanced framework and traditional Control-Flow Analysis. Concert also provides an advanced cloning mechanism [36].

4 Definition

This section presents a formal definition of object inlining in terms of program semantics, and a discussion of when it is semantics preserving.

4.1 Semantics

We first detail our semantic model of program execution, and then define the *execution trace*, a representation of entire

[1] A real inventory control program would likely use databases rather than linked lists. This example is for illustrative purposes only.

[2] Note that the transformation is not done source to source, this example is just for expository purposes.

```
class ItemListAndItem {
  ItemList *next;
  int product_id;
  int department;
  float value;

  Item(int p, int d, float v) {
    product_id = p;
    department = d;
    value = v; }

  ItemList(ItemList *d, ItemList *n)
    next = n; }

  double Item::value() { return value; }

  double ItemList::value() {
    if (!next)
     return this->Item::value();
    else
     return
      this->Item::value()+next->ItemList::();
  }
};

ItemList *inventory = NULL;

ItemList *AddItem(int id, int d, float v) {
  ItemListAndItem *l = new ItemListAndItem;
  l->Item(id, d, v);
  l->ItemList(item, inventory);
  inventory = l; }
```

Figure 2: Inventory Program (after Object Inlining)

program executions.

4.1.1 Semantic Model

Our semantic model has three elements: *state*, the dynamic and heap state at runtime; *nodes*, the basic actions performable by any program; and *regions* which group nodes according to their control dependence. A *program* consists of a set of methods, each of which has a set of nodes arranged in a tree of regions. All *values* in a program are write once, that is, the graphs are in SSA[3] form, with ψ and ϕ nodes associated with their governing conditionals. Our semantics defines the execution of such a program.

State There are two pieces of *state*, the bindings and the heap, which are each organized as a table indexed by an identifier (a variable name for binding state and an object name, field name pair for heap state) and a node. The node represents the *time* of the binding: indexing at a node n returns the most recent binding performed before n executes.

- A *binding* represents the value of local variables, i.e. it maps a variable name to a runtime value such as an integer or an object name. It is indexed by the name of the local variable and the time — i.e. the node — of the binding, as follows:

$$\mathfrak{B}(name, n_i)$$

- The global *heap* represents the state of all the allocated objects. State is essentially a global hash table $\mathfrak{H}$ mapping object name plus field name to runtime value. As with bindings, it is sensitive to time, i.e. the node being executed. The reference function takes an object name or local variable that is bound to an object, the field being referenced, and the statement being executed.

$$\mathfrak{H}(object, field, n_i)$$

Nodes Nodes represent the actions performed by programs. Nodes are defined by two features: *kind* denotes the action to be performed, e.g. a function call or an assignment. And *values* are the values being acted upon, e.g. the arguments to a function call, or the value being assigned. The values are mapped by the runtime state $\mathfrak{B}$ to runtime entities such as objects, integers, floats, booleans, characters. Figure 3 defines the semantics of each node kind (t will be discussed below); we use n_i to indicate the ith node to execute, which is needed because our state must be time-sensitive.

Regions Regions represent control flow; a region is defined as the set of nodes that execute under a given control condition. Thus, a region is the child of an `if` or a `while` node, and contains the nodes that execute under that conditional. Since conditional nodes are themselves in regions, a method is a tree of regions, rooted in a special region representing method entry. Within a region, the nodes execute subject to an ordering relation $\succ$ that represents data dependencies and any other orderings needed to satisfy program semantics. The semantics $\mathfrak{S}_\mathfrak{r}(r)$ of region r is shown in Figure 4.

```
done $\leftarrow$
**let** $\mathfrak{R} \leftarrow \{n \,|\, \nexists_{n_i} n_i \succ_r n\}$
**while** $\mathfrak{R}$ **do**
      **let** $n \in \mathfrak{R}$
      $\mathfrak{S}_\mathfrak{n}(n)$
      $\mathfrak{R} \leftarrow \mathfrak{R} - \{n\}$
      $done \leftarrow (done \cup \{n\})$
      **foreach** $n$ **in** $children(r)$ **do**
            **if** $\nexists_{n_i} ((n_i \succ_r n) \wedge (n_i \notin \text{done}))$
                  $\mathfrak{R} \leftarrow \mathfrak{R} \cup \{n\}$
```

Figure 4: Definition of $\mathfrak{S}_\mathfrak{r}(r)$: Semantics of Program Regions

4.1.2 Execution Traces

Execution traces allow us to reason about entire program executions. Execution is represented as the ordered sequence of nodes executed during a program execution, with actual runtime values bound to variables by $\mathfrak{B}$, and the ϕ and ψ nodes of SSA form turned into the appropriate moves. Since an execution trace represents the actions performed during an execution, conditional nodes are not included.

Definition 1 (Execution Trace). An *execution trace* e of program p is a triple $\langle\langle n_1, n_2, \ldots, n_i\rangle, \mathfrak{B}, \mathfrak{H}\rangle$, which consists of the following elements:

- $\langle n_1, n_2, \ldots, n_i\rangle$ is an ordered sequence of nodes representing the operations performed during the execution of p. Nodes in the sequence of trace e may be referred to as $\downarrow_e (k) = n_k$; the index of a node is $\uparrow_e (n_i) = i$

- $\mathfrak{B}$ represents the bindings during the execution; the bindings of a given trace e is referred to as $\mathfrak{B}_e$

[3] We actually use *Static Single Use* [34], a form of gated SSA such that any temporary is used in exactly one region.

kind	*form*	*semantics*
move	$\Vert\mathtt{x = y}\Vert$	$\mathfrak{B}(\mathtt{x}, n_{i+1}) \leftarrow \mathfrak{B}(\mathtt{y}, n_i), t \leftarrow t :: n_i$
write	$\Vert\mathtt{o.f = v}\Vert$	$\mathfrak{H}(\mathtt{o}, \mathtt{f}, n_{i+1}) \leftarrow \mathfrak{B}(\mathtt{v}, n_i), t \leftarrow t :: n_i$
read	$\Vert\mathtt{v = o.f}\Vert$	$\mathfrak{B}(\mathtt{v}, n_{i+1}) \leftarrow \mathfrak{H}(\mathtt{o}, \mathtt{f}, n_i) t \leftarrow t :: n_i$
send	$\Vert\mathtt{o.fun(}a_1, a_2, \ldots, a_j\mathtt{)}\Vert$	$\forall_v\ \mathfrak{B}(v, n_{i+1}) \leftarrow \bot$ $\forall_{1\le k\le j}\mathfrak{B}(\mathit{Dispatch}(\mathtt{o}, \mathtt{fun}, n_i)_{\rightarrow}(k), n_{i+1}) \leftarrow \mathfrak{B}(a_k, n_i)$ $\mathfrak{B}(\mathit{continuation}, n_{i+1}) = n_i$ $\mathfrak{S}_{\mathfrak{r}}(\mathit{entry}(\mathit{Dispatch}(\mathtt{o}, \mathtt{fun}, n_i)))$ $t \leftarrow t :: n_i$
return	$\Vert\mathtt{return\ v}\Vert$	$\forall_v \mathfrak{B}(v, n_{i+1}) \leftarrow \mathfrak{B}(v, \mathfrak{B}(\mathit{continuation}, n_i))$ $\mathfrak{B}(\mathfrak{B}_e(\mathit{continuation}, n_i)_{\leftarrow}(0), n_{i+1}) \leftarrow \mathfrak{B}(\mathtt{v}, n_i)$ $t \leftarrow t :: n_i$
if	$\Vert\mathtt{if\ (v)}\Vert$	$\mathtt{if}\ \mathfrak{B}_e(v) \neq 0$ $\forall_{p=\Vert\psi(\mathtt{v_t},\ \mathtt{v_f})\ =\ \mathtt{v}\Vert\in psi(n_i)}\mathfrak{B}(v_t, n_{i+1}) \leftarrow \mathfrak{B}(\mathtt{v}, n_i), t \leftarrow t :: \Vert v_t\ \mathtt{=}\ \mathtt{v}\Vert$ $\mathfrak{S}_{\mathfrak{r}}\mathit{child}(n_i, \mathit{true})$ $\forall_{p=\Vert\mathtt{v}\ =\ \phi(\mathtt{v_t},\ \mathtt{v_f})\Vert\in phi(n_i)}\mathfrak{B}(\mathtt{v}, n_i) \leftarrow \mathfrak{B}(v_t, n_{i+1}), t \leftarrow t :: \Vert\mathtt{v}\ \mathtt{=}\ v_t\Vert$ $\mathtt{else}$ $\forall_{p=\Vert\psi(\mathtt{v_t},\ \mathtt{v_f})\ =\ \mathtt{v}\Vert\in psi(n_i)}\mathfrak{B}(v_f, n_{i+1}) \leftarrow \mathfrak{B}(\mathtt{v}, n_i), t \leftarrow t :: \Vert v_f\ \mathtt{=}\ \mathtt{v}\Vert$ $\mathfrak{S}_{\mathfrak{r}}\mathit{child}(n_i, \mathit{false})$ $\forall_{p=\Vert\mathtt{v}\ =\ \phi(\mathtt{v_t},\ \mathtt{v_f})\Vert\in phi(n_i)}\mathfrak{B}(\mathtt{v}, n_i) \leftarrow \mathfrak{B}(v_f, n_{i+1}), t \leftarrow t :: \Vert\mathtt{v}\ \mathtt{=}\ v_f\Vert$
while	$\Vert\mathtt{while\ (v)}\Vert$	$\forall_{p=\Vert\mathtt{v}\ =\ \phi(\mathtt{v_t},\ \mathtt{v_f})\Vert\in phi(n_i)}\mathfrak{B}(\mathtt{v}, n_i) \leftarrow \mathfrak{B}(v_f, n_{i+1}), t \leftarrow t :: \Vert\mathtt{v}\ \mathtt{=}\ v_f\Vert$ $\mathtt{while}\ \mathfrak{B}_e(v) \neq 0$ $\mathfrak{S}_{\mathfrak{r}}\mathit{child}(n_i, \mathit{true})$ $\forall_{p=\Vert\mathtt{v}\ =\ \phi(\mathtt{v_t},\ \mathtt{v_f})\Vert\in phi(n_i)}\mathfrak{B}(\mathtt{v}, n_i) \leftarrow \mathfrak{B}(v_t, n_{i+1}), t \leftarrow t :: \Vert\mathtt{v}\ \mathtt{=}\ v_t\Vert$

Figure 3: Definition of $\mathfrak{S}_{\mathfrak{n}}(n_i)$: Execution Semantics of Program Nodes. The extended semantics for trace generation $\mathfrak{T}_{\mathfrak{n}}(n_i)$ are shown by operations on t.

- $\mathfrak{H}$ represents the heap during execution; the heap of a given trace e is referred to as $\mathfrak{H}_e$.

The state in an execution trace represents the state built by the semantics of Section 4.1.1, and, in that figure, the node sequence is represented by t, with each rule extending t as appropriate.

4.2 Object Inlining

Object inlining replaces uses of child objects with uses of corresponding parents. It is well-defined only for what we call a one-to-one field, which we define below. Then, we define what object inlining actually does at execution time, and then claim and prove the conditions under which it is semantics preserving.

Definition 2 (Dynamic One-to-One Field). A field f is a *dynamic one-to-one field*, written $\xrightarrow{f}$, for a given execution of program p if every parent object corresponds to exactly one child object thru that f. This is defined formally as follows:

$$(n_1 = \Vert o_1\mathtt{.f = }\ v_1\Vert \wedge n_2 = \Vert o_2\mathtt{.f = }\ v_2\Vert) \rightarrow \\ (\mathfrak{B}(o_1, n_1) = \mathfrak{B}(o_2, n_2) \leftrightarrow \mathfrak{B}(v_1, n_1) = \mathfrak{B}(v_2, n_2))$$

Definition 3 (Dynamic Field Inlining). A *dynamic field inlining* $\mathfrak{I}^f$ of f is defined if and only if f is a dynamic one-to-one field for a given execution. It involves substituting corresponding parent objects for child objects wherever they appear in the execution. This is formalized as applying following transformation to all values in the binding and heap environments for each node that executes.

$$\mathfrak{I}^f(v) \leftarrow \begin{cases} \mathfrak{B}(\mathtt{p}, n) & ((n = \Vert\mathtt{p.f = c}\Vert) \wedge (v = \mathfrak{B}(\mathtt{c}, n))) \\ v & \mathit{otherwise} \end{cases}$$

Altering object references at random in a given program does not, in general, preserve the meaning of that program. However, dynamic field inlining does preserve program semantics when it is applied to a one-to-one field, as we prove below.[4]

Lemma 1 (Dynamic field inlining preserves sharing). *Sharing patterns in references to child objects are unchanged when dynamic field inlining is applied to a dynamic one-to-one field. That is, two child state access nodes reference the same object field after the transformation if and only if they did so beforehand. More formally:*

$$\forall_f\ \forall_{n_1=\Vert v_1\ \mathtt{=}\ o_1\mathtt{.f}\Vert, n_2=\Vert o_2\mathtt{.f}\ \mathtt{=}\ v_2\Vert\in e_p} \\ \mathfrak{I}^f\left(\mathfrak{B}\left(o_1, n_1\right)\right) = \mathfrak{I}^f\left(\mathfrak{B}\left(o_2, n_2\right)\right) \Leftrightarrow \mathfrak{B}\left(o_1, n_1\right) = \mathfrak{B}\left(o_2, n_2\right)$$

Proof. Details omitted, but follows directly from Definitions 2 and 3, since $\mathfrak{I}^f$ is a one-to-one mapping except that it maps pairs of parent and child objects both the parent. Since a one-to-one mapping preserves equality, Lemma 1 is clearly correct unless o_1, o_2 are a parent, child pair, but that cannot be the case, as they could not share `f`, due to our α-renaming. □

[4] Since our model of state uses an object name, field name pair as a key, merging parent and child objects that share a field name would cause ambiguity when naming heap storage. Before object inlining, we first perform an α-renaming to prevent such name collisions.

Theorem 1 (Dynamic Inlining State Consistency). *After dynamic field inlining $\mathfrak{I}^f$ of f, all reads of any object field return the value of the dynamically previous write to that object field. More formally,*

$$\forall_{\{n_i | n_i = \|\texttt{v = c.x}\|\}} \mathfrak{I}^f_v(\mathfrak{B}(v, n_{i+1})) = \mathfrak{I}^f_v(\mathfrak{H}(c, x, n_i))$$

Proof. By Lemma 1, for any field access $x.f$, exactly the same sets of statements read and write that field after $\mathfrak{I}^f$ as did so before it. Thus, the same value v_s was dynamically last assigned into $x.f$ after $\mathfrak{I}^f$ as before it. Also, the bindings of v_s, v were both transformed by $\mathfrak{I}^f$, so, given that they were the same before, for that is the semantics of a field read, they must also be the same afterward. Hence, state access is still consistent. □

Theorem 2 (Inlined Field Redundancy). *After dynamic field inlining $\mathfrak{I}^f$, in any statement of the form $n = \|\texttt{v = o.f}\|$ before dynamic field inlining, $v = o$. That is to say*

$$\forall_{\{n | n = \|\texttt{v = o.f}\|\}} \mathfrak{B}_e(v, n) = \mathfrak{B}_e(o, n)$$

Proof. By Definition 3, all statements of the form $\|\texttt{o.f = c}\|$ get transformed into $\|\texttt{o.f = o}\|$, since it is exactly such statements that *define* the mapping $\mathfrak{I}^f$. Then, by Theorem 1, for all statements of the form $\|\texttt{v = o.f}\|$, $\mathfrak{B}_e(v, n) = \mathfrak{B}_e(o, n)$. □

By Lemma 1 and Theorem 1, we have that the program is observationally equivalent after object inlining is applied to a one-to-one field, and by Theorem 2, the inlined field is no longer needed. Hence, by removing that field, we get a program that does the same thing but does not have the extra reads and writes for the inlined field.

5 Analysis

We present a compiler analysis that identifies safely inlinable object fields in two steps: first, we must prove that a candidate field for inlining is one-to-one. Second, we must precisely find all program points where child objects are used, so they can be redirected to their corresponding parent. The analysis for both problems appears simple, but depends critically on the context-sensitive and data-sensitive iterative inter-procedural analysis provided by the Concert system (mentioned in Section 3.). Without the advanced capabilities of the Concert system, we would be unable to conveniently express the complex analysis, precisely refine and track the data and control flows as required, and do so with practical efficiency.

5.1 Determining One-to-One Fields

Our algorithm to prove a field f associated parents and children in a one-to-one manner consists of two steps: 1) verifying that the given parent and child object creations and a corresponding field always execute together, and 2) checking that created objects assigned by the corresponding assignment.

5.1.1 Verifying Common Execution

Common execution ensures both parent and child are created together, and is ensured if the parent and child creation nodes exist in the same inter-procedural control dependence region (ip-region). These ip-regions generalize traditional control dependence regions[5, 15] to be inter-procedural. More precisely, an ip-region is governed by a conditional, and its members are a set of <node, edge stack> pairs where nodes are the program nodes dependent on the governing conditional, and the edge stack is the method calls between the conditional and the node. Because nodes can be in a method which is called from many places, nodes can in general belong to multiple ip-regions or to the same region more than once. However, they will have a different edge stack in each ip-region. Figure 5 shows pseudocode for computing the set of ip-regions to which a node belongs.

```
CalcRegions(n)
    ; parent is control parent of node
    if $parent(n)$ is conditional then
        ; governing conditional determines ip region
        return $parent(n)$
    else
        ; must be in entry region of function
        ; so use the ip regions of the callers
        foreach $c$ in $callers(method(n))$ collect
            ; record governing conditionals and call paths
            $c :: CalcRegions(c)$
```

Figure 5: Algorithm to Compute Node's IP Region

If the creation nodes of the parent and child objects exist in the same ip-region, then we have verified their common execution.

5.1.2 Verifying Data Flow

Verifying data flow ensures that the parent and child objects flow unambiguously to the assignment of the child to field f. Verification involves checking that the values at the creations and at the assignment must be aliased, for which any of a number of alias analysis algorithms [30, 14, 37, 32] can be applied. We exploit the iteratively refined inter-procedural analysis framework of Concert, and simply follow data flow between creations and assignments, checking that there are no merges, as shown algorithmically in Figure 6. Upon request, the Concert system iteratively refines the analysis results, expending effort only along the paths of interest, and while they are still potentially unaliased.

```
CheckNoMerges($v_1$, $v_2$)
    ; all $v$ such that $v_1$ is the only data predecessor ($\succ$ indicates data flow)
    let $V \leftarrow \left\{ v \,\middle|\, \begin{array}{l} v = v_1 \vee \\ \left( \exists v_i \begin{array}{l} v_i \succ v \wedge \\ v_i \in V \wedge \\ \nexists v_j \left( \begin{array}{l} v_j \succ v \wedge \\ v_j \neq v_i \end{array} \right) \end{array} \right) \end{array} \right\}$
    return $v_2 \in V$
```

Figure 6: Algorithm to Check for Data Flow Merges

5.1.3 Computing One-to-One Fields

One-to-one fields are computed with the two algorithms above as shown in Figure 7. If a field satisfies the one-to-one and precise data-flow criteria, it is a safe candidate for inlining. The code below is slightly simplified: there is also a check that, for a given one-to-one pair of creations for a field f, no other creations get assigned into f for the given parent creation.

```
CheckOneToOneField($f$)
; check each assignment to $f$
foreach $a$ is ||o.f = v|| do
    ; check each ip-region of the creators of o
    ; $Creators$ are possibly creating news
     foreach $p$ in $CalcRegions(Creators($o$))$
        ; find corresponding child creation, if any
         unless $find\ p\ CalcRegions(Creators($v$))$
            return $false$
    ; check each ip-region of the creators of v
     foreach $c$ in $CalcRegions(Creators($v$))$
        ; find corresponding parent creation, if any
         unless $find\ c\ CalcRegions(Creators($o$))$
            return $false$
return $true$
```

Figure 7: Checking for One-to-One Fields

5.2 Finding Field Uses Precisely

Given one-to-one fields, a safe inlining transformation requires precise identification of all uses of the child. Our data-flow analysis is conceptually straightforward, but depends heavily on the Concert infrastructure to provide precise data flow paths at acceptable cost. Essentially, we track data flow from the parent and child at each one-to-one field assignment back to their respective creation points. From there, we track data flow forward to all uses of the child object fields.

Definition 4 (Tags). In the actual implementation, we three types of value *Tags* as indicated, and are propagated as shown in Figure 8.

Forward Tags The *forward tags* of a value v, written $\mathfrak{T}^{\rightarrow}(v)$, tag child field reads with the parent

Backward Tags The *backward tags* of a value v, written $\mathfrak{T}^{\leftarrow}(v)$, tag creation nodes with parent

Forward Backward Tags The *forward backward tags* of a value v, written $\mathfrak{T}^{\leftrightarrow}(v)$, tag child field reads and writes with parent (before object is assigned to the parent)

The algorithm collects all of the child object uses which are forward from the child to parent assignment using the forward tags, and all of the child object uses which are forward from the child object creation point to the child to parent assignment using the forward-backward tags. This is all of the uses of the child object.

6 Transformation

Given inlinable fields, we perform the object inlining transformation in three steps:

```
 CheckPreciseUses($f$)
; backward tags: all values that must flow to $f$
 let $B \leftarrow \{v\,|v = f \vee (\exists v_i \in B\ v \succ v_i \wedge \nexists v_j\ (v \succ v_j \wedge v_j \neq v_i))\}$
; forward tags: all values to which $f$ might flow
 let $F \leftarrow \{v\,|v = f \vee \exists v_i\ v_i \in F \wedge v_i \succ v\}$
; forward backward tags: all values to which $B$ might flow
 let $FB \leftarrow \{v\,|v \in B \vee \exists v_i\ v_i \in F \wedge v_i \succ v\}$
; found precisely in set closed under $\succ$
 let $U \leftarrow (F \cup B \cup FB)$
 return $\nexists v_i, v_j\ (v_i \succ v_j \wedge v_j \in U \wedge v_i \notin U) \wedge$
 $\nexists v_i, v_j\ (v_j \succ v_i \wedge v_j \in U \wedge v_i \notin U)$
```

Figure 8: Checking for Precise Uses of Fields

1. Build new fused classes based on inlinable fields
2. Replace object creations with fused object creations
3. Replace child state accesses with fused object accesses

With the results of the analysis described in Section 5, Steps 2 and 3 are straightforward, since the analysis precisely computes exactly the sets of creations and uses to transform. So, we focus on building the new fused classes.

6.1 Building Fused Classes

Given the original classes, object inlining identifies a set of inlinable fields. The inlining of those fields defines a new set of classes, where each inlining operation collects the fields of two classes into a new fused object class. For instance, the example in Section 2 collected the fields from `Item` and `ItemList` into a single new object class, `ItemListAndItem`.

Choosing object layouts in the fused classes is critical as it determines the code size increase that object inlining incurs. The fused object classes combine data from distinct source program class definitions, and must also combine the methods associated with those source program classes. If the methods are to be shared, the layout of the fields must conform (i.e. be at the same location in the object) in all the classes in which the fields appear. If not, the methods must be cloned, increasing the program code size.[5]

We choose object layouts by the algorithm shown in Figure 10, using an approach that roughly speaking tries to collect the fused object classes into trees (analogous to single inheritance hierarchies). If the classes can be organized into a tree—where nodes are classes, and children contain supersets of fields in their parent—then a conformant layout can then be obtained by doing a pre-order traversal of the tree with respect to each class. Our strategy for laying out inlined classes is based on this observation, and we diverge from this approach only when non-conformance is required.

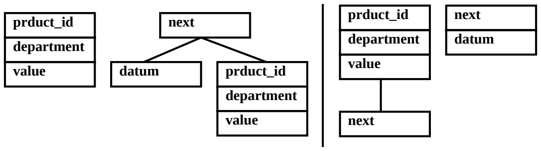

Figure 9: The Possible Layouts of Example

The key observation is that non-conformance can only arise when two fields appear in overlapping subsets of classes, neither being a subset of the other. This induces two distinct sets of constraints on the field placement in the

[5] This is in part because we use a JVM-like object reference model which forbids internal pointers.

object (as the sets of classes could be connected to the hierarchy in two places). In such cases, we replicate the field, which implies cloning all methods which access that field, and attach the subsets to distinct parts of the class hierarchy. There remains a subtle choice for where to attach the classes in the overlap region, where the choice determines which of the two non-overlapping subsets the overlap region will conform with. This choice is illustrated in Figure 9, where there are two choices for laying out the parent, child and fused classes from Section 2. Because the cost of cloning is not known until we have all object layouts, we do not have enough information to choose well, and defer this choice, constructing both alternatives.

```
; input: each $set \in sets$ is a field and the classes it is in
; output: a forest denoting the layouts of all classes
FindLayout(sets)
; current set of classes
let $classes \leftarrow \bigcup_{s \in sets} Classes(s)$
; find fields that appear in all classes
let $top \leftarrow \{s \,|\, s \in sets \wedge Classs(s) = classes\}$
; if any, put at the front of the layout
if $|top| \neq 0$ then
        return $MakeTree(top, FindLayout(sets - top))$
else
      ; builds sets of fields that appear together in some class
      let $sets \leftarrow \{\{s_1, s_2, \ldots\} \,|Classes(s_1) \cap Classes(s_2)\}$
      ; disjoint fields laid out independently
      if $|sets| > 1$ then
            return $MakeForest(map\ FindLayout\ sets)$
      ; otherwise, there is a conflict
      else
            ; $s_1, s_2$ cannot be laid out conformantly
            let $s_1, s_2$ such that $Classes(s_1) \cap Classes(s_2) \wedge Classes(s_1) - Classes(s_2) \wedge Classes(s_2) - Classes(s_1)$
            ; three choices to resolve conflict
            let $cut \leftarrow \left\{ \begin{array}{l} Classes(s_1) - Classes(s_2) \\ Classes(s_2) - Classes(s_1) \\ Classes(s_1) \cap Classes(s_2) \end{array} \right.$
            ; find layouts after splitting conflicting fields
            return $FindLayout(sets - \{s_1, s_2\} \cup \{s_1 - cut, s_2 - cut, s_1 \cap cut, s_2 \cap cut\})$
```

Figure 10: Layout Classes to Minimize Non-Conformance

The class layout algorithm produces a set of candidate layout trees, from which we must choose a single one. To do so, we calculate the cloning cost of each layout, and take the one which produces the fewest method clones (smallest code size increase). The amount of cloning required is determined by the complete for each class, the actual use of the non-conformant fields, and the specific cloning and method dispatch mechanisms.

7 Evaluation

We evaluate the effectiveness of object inlining, presenting compile-time and runtime results for a range of object-oriented program types and sizes.

7.1 Metrics

We present compile-time and runtime metrics to assess object inlining. compile-time metrics measure analysis power, analysis cost, and cloning cost. Runtime metrics measure analysis power, field reads, field writes, object allocations, and overall running time.

Analysis Power Analysis power is the number of inlinable fields, which we measure at both compile- and run-time. At *compile time*, we count the number of declared object fields found to be inlinable by our analysis. Partially-inlinable fields are counted fractionally, e.g. a field inlinable in 3 of 4 cases is 0.75 fields. We also present this number normalized to the number of declared fields in the programs, to provide calibration and a measure of scaling; we also calibrate by comparing with the number of fields declared inline in our C++ benchmark programs. At *runtime*, we assess analysis power against our semantic criteria for inlinable fields. We present two dynamic ratios: the first is the number of compile-time inlinable fields—the sum of the number of compile-time inlinable fields over all allocated objects—normalized to the ideal number of inlinable fields—the sum over all allocated objects of the number of fields that satisfy our inlinability semantics; this gives the fraction of all inlinable objects that our analysis found. The second is the number of reads of inlinable fields divided by the number of reads of ideally inlinable fields; this gives the fraction of the pointer read overhead our analysis can remove.

Analysis Cost Analysis cost covers the amount of work done and size of data structures needed by the analysis system. In our constraint-based analysis system, the best unit of work is constraint propagations and for data size we use the common contexts per method metric. These are normalized by program size, measured as the number of values and methods in the internal form, to assess scalability. In order to place the power and cost of our analysis in perspective, we compare it with the traditional 1-CFA analysis framework.

Cloning Cost Code specialization required by object inlining is measured by counting method clones required. We normalize by the number of methods.

Field Reads and Writes Object inlining should reduce the number of object dereferences required to execute the program. We measure field reads directly and total loads, data cache misses and read stall cycles using hardware performance counters, capturing effectiveness in reducing object dereferences.

Object Allocations Object inlining should reduce the number of objects allocated. This metric captures the dynamic counts of allocation operations.

Program Runtime Object inlining should improve program runtime by improving program memory behavior. We measure overall runtime, and also the number of instructions executed, to analyze where performance improvements come from.

7.2 Benchmarks

We selected object-oriented programs and libraries for evaluation; we want a range of data structures and programming idioms, and a range of program sizes. To ensure this, we settled upon three class libraries, the SmallTalk-like NIHCL, the more static OATH, and AI, containing various intelligent search routines. All three libraries have test applications. We also choose a range of object-oriented programs. These benchmarks are summarized in Table 1. We break out application and library sizes because, even though our whole-program compiler makes no distinction, much library code is unused in a given application, and hence the amount of live code is much less than a simple sum of program and library size would suggest.

benchmark	lines	inlinable data structures
oopack	760	complex numbers
AI library (3000 lines) programs		
demo3	300	lists, search nodes, arrays
demo4	300	lists, search nodes, arrays
demo6	200	lists, arrays
NIHCL library (20000 lines) programs		
bag	100	bag, set, array, iterator
dict	100	assoc, bag, set, array, iterator
orderedcltn	100	collections, array, iterator
sets	100	set, array, iterator
OATH library (18000 lines) programs		
dll0	150	arrays, smart pointers
large programs		
otest	30000	lists, arrays, wrappers, parser
xpdf	25000	streams, arrays, child

Table 1: Benchmarks with Inlinable Data Structures

7.3 Experimental Setup

We compiled each benchmark with and without object inlining for both our adaptive analysis and traditional 1-CFA using the Concert Compiler, and the Concert-generated C++ code with g++ (egcs-2.91.66) with full optimization (-O3). These two options are labelled 'inlining' and 'base' in our results. For calibration, we compiled the original C++ codes with g++ -O3, giving g++ the whole program at once. We used a 266MHz Pentium with 64MB of RAM, running RedHat Linux 6. Our low-level measurements used the PMC software from NASA Ames.

Since the benchmark programs are C++ codes, they typically have fields manually inlined. Since Concert supports a Java-like uniform reference model, all objects are accessed via pointers, even if declared inline allocated. Concert generates appropriate code to preserve the original C++ semantics using a reference model. Concert with object inlining fuses multiple objects together, preserving the uniform reference model implementation even when inline allocating objects.

7.4 Calibration Results

To provide calibration for the performance of the Concert compiler, with and without object inlining, we compiled our benchmark programs using g++ and compared the performance to that of the programs compiled by Concert. Figure 11 presents Concert performance normalized to G++, and shows competitive performance.

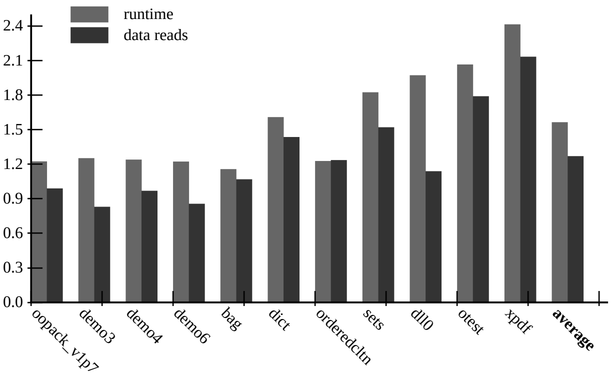

Figure 11: Performance comparison of Concert and G++

7.5 Object Inlining Analysis Results

Analysis Power. Table 2 shows our compile-time analysis results; most programs have a surprisingly large number of inlinable fields: typically more than one third of fields are inlinable, and the result scales almost uniformly with program size. The total number of fields with object type is the left-hand column labelled "All." There are four columns labelled "Inlinable": the first (labelled "Hand") is the number of fields explicitly inlined in the C++ source code, the second is a count of the automatically inlinable fields as found by our analysis using Concert's advanced framework, the third (labelled "1-CFA") is the number of fields found inlinable using our analysis in a traditional 1-CFA analysis framework, and the fourth (labelled "%") is the number of inlinable fields found by our analysis as a percentage of the total fields. Our adaptive analysis always does better, usually much better, than 1-CFA; in addition, in all cases except *xpdf*, it finds substantially more inlinable fields than were declared by hand in C++. These results are significantly better than our previous ones, for example *otest* has 15.7 inlinable fields compared to 9 in our prior study [20]. Fields that can be inlined in some cases but not in others are counted as a fraction, based on the number of situations in which they can be inlined.

	Fields				
		Inlinable			
Benchmark	*All*	*Hand*	*Precise*	*1-CFA*	*%*
oopack_v1p7	4	0	0.5	0	12
demo3	13	3	5	0	38
demo4	14	3	6	0	42
demo6	14	5	6	0	42
dll0	17	2	2.74667	1	16
bag	30	6	11.2022	8.5	37
dict	31	6	11.4189	6.5	36
orderedcltn	27	4	10.4732	7.5	38
sets	27	4	10.381	7.5	38
otest	38	5	15.7167	10.5	41
xpdf	133	54	45.25	23.4348	34

Table 2: Automatically Identified Inlinable Object Fields

Figure 12 assesses the effectiveness of our compiler analysis at finding semantically inlinable fields; since they are based on program semantics, these measurements are of counts of dynamic objects at runtime. The graph shows two bars for each program: the first bar is the fraction of all objects that satisfy the semantic criteria for inlinability that are found to be inlinable by our analysis. This measures directly how good an approximation of our inlinability semantics our analysis is. The second bar shows the fraction of all possible reads of inlinable fields that our static analysis finds; this is simply the number of reads of statically inlinable fields as a fraction of the number of reads of ideally inlinable fields. The results vary, and are generally better for the small programs. But even for the large programs, the fraction is often more than one third, with the averages across all programs being roughly 40%.

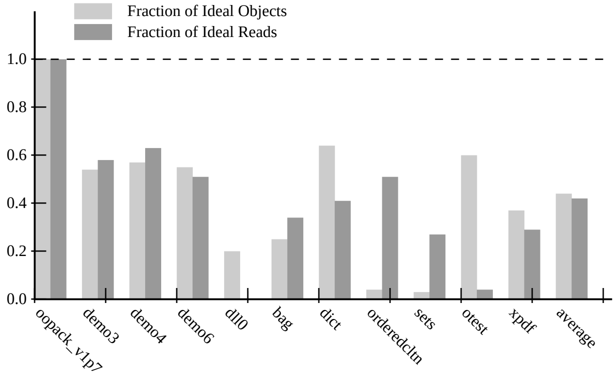

Figure 12: Fraction of Semantic Ideal for Static Analysis

Analysis Cost. Figures 13 and 14 reveal that our object inlining optimization has reasonable analysis cost, and that cost does not increase rapidly with program size (as normalized by the number of values in the program source). Overall, the cost of our analysis is less when using Concert's advanced analysis framework than when traditional 1-CFA, despite getting better results, and the Concert framework scales with program size much better than 1-CFA. We believe that Concert's demand-driven analysis system allows practical scalability to much larger programs. Figure 13 counts average contexts per method, for our adaptive analysis and 1-CFA. This captures the space requirements of our analysis. All of the numbers for our analysis are relatively low, with the maximum being about 2.4 contexts per value for *xpdf*. There is no discernible trend of cost with program size for our analysis; indeed, the largest program, *xpdf*, has one of the lowest costs. 1-CFA, on the other hand, shows dramatic cost growth for the largest programs, *otest* and *xpdf*.

Figure 14 shows the total work done by the analysis system for our analysis in both Concert and 1-CFA frameworks; it shows the number of constraint propagations normalized by program size. As our program analysis is constraint-based, constraint propagations capture amount of work the analysis system does. Concert's demand-driven refinement allows significant directed analysis which can follow the critical program structure, but allows it to be done at reasonable cost. Overall, the amount of work does grow as program size grows, although the largest program, *xpdf*, has one of the lowest costs. The dominant effect is of program structure, and the need for analysis to unravel it. For example, *dict* has an especially complex structure, with mutually recursive assignments thru state, and that causes it to have considerably more work than any other program. For 1-CFA, cost once again grows substantially for the largest program, *xpdf*.

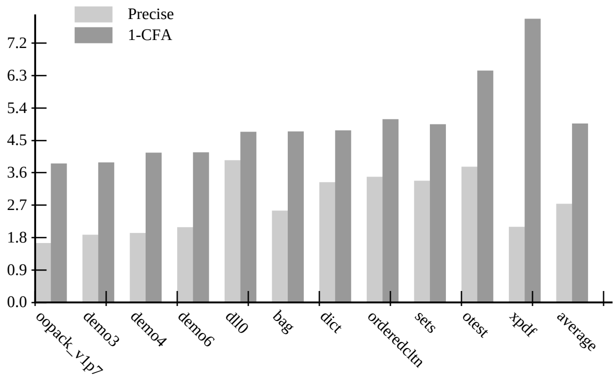

Figure 13: Context Sensitivity required for Object Inlining. Analysis data size to find inlinable fields.

Since, as we have seen above, our analysis finds strictly more fields than 1-CFA, and usually many more, at a competitive or lower cost, we shall present the rest of our results of our own adaptive analysis only.

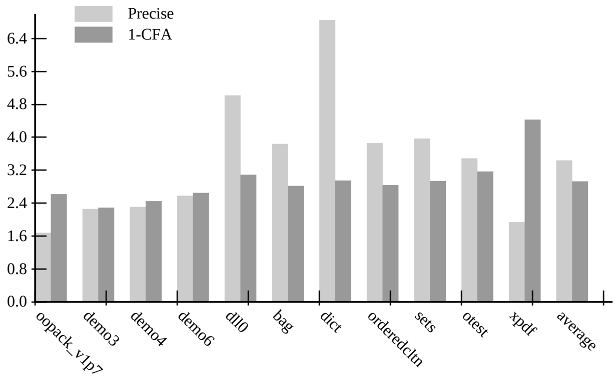

Figure 14: Analysis Cost. Shows analysis computational to identify the inlinable fields (normalized to program size).

Cloning Cost. Figure 15 shows that object inlining has modest and scalable cloning requirements; the number of required clones is below 1.5 per method for all programs except *demo6*, and the second largest code, *otest*, has the smallest number of required clones, at 1.04 per method. These small numbers are despite polymorphic inlining in which the same class is inlined into multiple fields; the number of clones is small because our object layout algorithm is able to arrange that almost all of these cases have conformant object layout.

7.6 Object Inlining Impact Results

Field Accesses Figures 16 and 17 and show that object inlining substantially reduces the dynamic count of field accesses. This yields significant improvements in memory performance. Figure 16 has four bars for each program; from left to right, they are object field reads, all program data loads, L1 data cache misses, and data dependence induced read stalls. The numbers show the fraction of operations from the base program remaining in the inlined one. Field reads are reduced by 4% to 46% with an average reduction of 28%. Data loads are reduced from 3% to 26%, with an

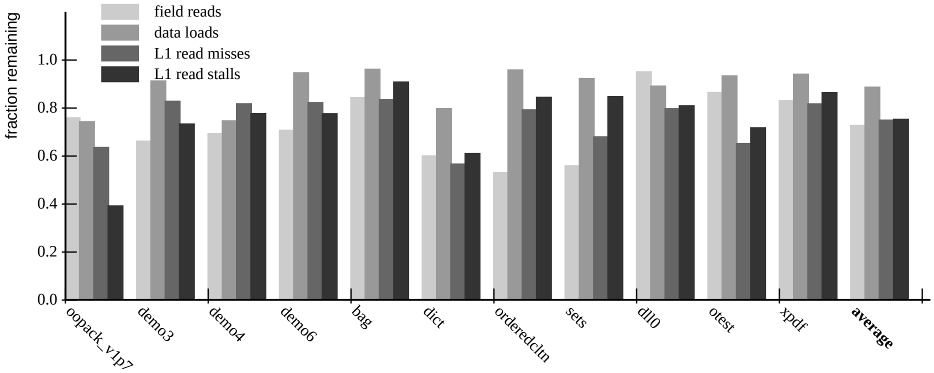

Figure 16: Field Read Counts and L1 cache misses with and without Object Inlining

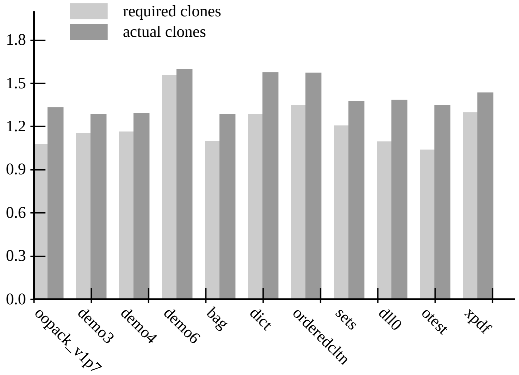

Figure 15: Cloning Costs of Object Inlining. Shows the number of additional methods which must be cloned as a result of the object inlining optimization.

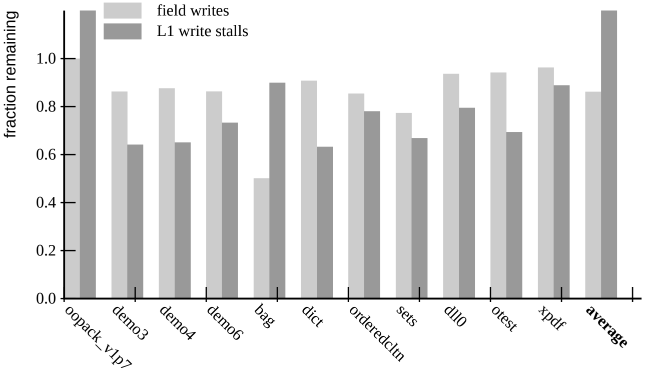

Figure 17: Field Write Operations with and without Object Inlining. Shows the significant benefits of object inlining on field writes.

average reduction of 12%. Data cache misses and read stalls are reduced by an average of 25%.

Figure 17 shows the reductions in memory writes due to object inlining. It has two columns: on the left is the fraction of field writes remaining after object inlining, and on the right is the fraction of write stall cycles remaining. The fraction of writes removed is 15% and the fraction of write stalls removed averages 27% excluding `oopack` which is anomalous and increases by 12 times due to removing the heap accesses from a tight loop, causing the remaining writes to run much faster. However, the overall program runtime does improve.

Object Allocations Figure 18 shows that object inlining greatly reduces the number of objects allocated: it removes nearly half of all objects in all cases but one, and removes an average of 58%. Furthermore, the number of bytes allocated drops by 35% on average as well, due to the reduced object overhead of fewer objects and the space saved by elided pointer fields. These results are significantly better than our previous results, which showed about 40% of objects being removed on a suite of smaller programs.

Program Runtime The improvements in memory system performance due to object inlining make a significant improvement in overall program performance, as shown in Figure 19. Figure 19 has two bars per program: the first is the instruction count and the second is runtime, both normalized to the program without object inlining. The reduction in instruction count is generally less than that in runtime, and the difference is explained by the reductions in read and write stalls shown above.

Our empirical results indicate that object inlining is successful in its objectives of reducing field reads and object allocations, removing 28% and 58% respectively. This, in turn, reduces cache misses by nearly one-third. And object inlining does make a significant improvement in overall program behavior: data reads are reduced by 12% and runtime by 14% on average.

8 Related Work

Our Previous Work. The idea of doing inline allocation automatically is by no means new [6], but previous systems have had insufficient analysis power achieve general automatic object inlining. For example, [8] presents a transformation that inlines the object state in stack frames, but they do not present automatic analysis sufficient to make it fully automatic. We introduced a basic object inlining analysis and optimization in [19] and presented a more extensive evaluation of it in [20]. Here we present a formal semantics for the analysis which we used to eliminate several limitations of the previous work. In [19], we described a simple mechanism for object layout which did not handle all cases, and caused a lot of cloning, particularly in large programs.

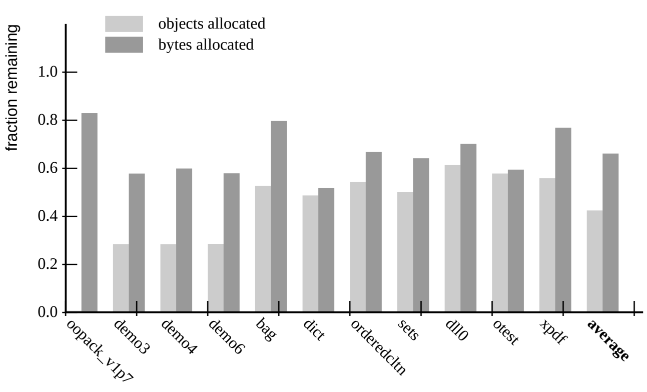

Figure 18: Object Allocations with and without Object Inlining. Shows how object inlining can dramatically reduce the number of objects which must be allocated.

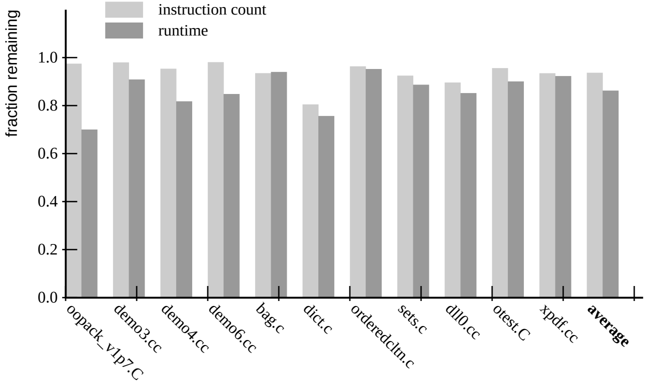

Figure 19: Program Runtime with and without Object Inlining. Shows significant overall runtime improvements based on object inlining.

In this paper, we have described an algorithm which globally optimizes the object layouts, keeping code size increases low. Finally, here we present a more comprehensive evaluation with much larger programs. This evaluation presents detailed low-level architecture data such as cache misses to more fully explore the operation of object inlining in practice. Finally, our overall results are significantly better than previous studies, including improvements the analysis effectiveness and impact on performance.

Object Oriented Systems. The impetus for object inlining is to match the performance of languages with explicit inlining like C++ [22] and Oberon-2 [43]. While we reap much of the performance benefit of custom object layouts, our approach allows programmer to write programs with a simple uniform object model, with compiler to do the data structure customization depending on use.

Recently, [12, 13], Chilimbi et al. presented several techniques for optimizing layout and allocation of objects to improve cache performance. Specifically, reordering fields within classes, controlling the co-allocation of objects, and splitting single objects to concentrate "hot" fields. Our optimizations are most similar to Chilimbi's field and object organization work, but our object transformation does not use profile information, and focuses on putting entire objects together, rather than splitting out the cold parts. Our work is quite different philosophically from the runtime co-allocation and object transformation work of Chilimbi which requires detailed programmer information for correctness. In contrast, our transformations are fully automatic, requiring no programmer intervention.

Unboxing. Much work on optimization of heap objects has focused on recursive data structures. In contrast, the primary source of object inlining found by our analysis is static indirection chains and pointer hierarchies induced by code reuse, modules, and generic class libraries. For example, optimization of recursive data structures in the functional programming community, *unboxing* (e.g. [25]), generates specialized representations to elide pointer dereferences. The analyses used for unboxing polymorphic structures resemble our analysis to track object uses, but the lack of state update simplifies the analysis problems. Shao et al. [38] present an analysis for unrolling linked lists, but only for the functional subset of ML – inline allocating tail pointers to some fixed depth.

Fortran. Optimizing array layout for cache performance [2, 44, 26] also involves transforming data layout. Fortran arrays are structurally immutable, which provides the same simplifications as are found in functional languages. Hence, the challenges in analyzing heap allocated object structures where data flow and aliasing are the major problems is significantly different.

9 Conclusions

We have presented further developments to the object inlining optimization we presented in [19, 20]: we developed a formal model of object inlining, and derived from that correctness conditions for object inlining; we presented an algorithm for laying out the fused classes that object inlining creates; and we carried out a more thoroughgoing evaluation of object inlining, measuring a range compile-time and runtime characteristics on a wider range of programs than before. Our results indicate that our newly-formulated analysis and transformation perform well and scalably—finding about 30% of fields inlinable—at moderate cost. We found our analysis had both better results and lower and more scalable cost than a traditional 1-CFA analysis. And the runtime impact is significant: on average, 28% of field reads, 58% of object creations, 12% of all loads, 25% of all L1 data cache misses and 25% of read stalls are removed. This improved runtime by an average of 14%.

Acknowledgements

The research described is supported in part by DARPA orders #E313 and #E524 through the US Air Force Rome Laboratory Contracts F30602-99-1-0534, F30602-97-2-0121, and F30602-96-1-0286. It is also supported by NSF Young Investigator award CCR-94-57809 and NSF EIA-99-75020. It is also supported in part by funds from the NSF Partnerships for Advanced Computational Infrastructure – the Alliance (NCSA) and NPACI. Support from Microsoft, Hewlett-Packard, Myricom Corporation, Intel Corporation, Packet Engines, Tandem Computers, and Platform Computing 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] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and computation transformations for multiprocessors. In *Proceedings of Fifth Symposium on Principles and Practice of Parallel Programming*, 1995.

[3] A. Ayers, R. Gottlieb, and R. Schooler. Aggressive inlining. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–145, Las Vegas, Nevada, June 1997.

[4] H. Baker. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[5] T. Ball. What's in a region? computing control dependence regions in linear time and space. Technical report, University of Wisconsin–Madison, 1992.

[6] 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.

[7] B. Blanchett. Escape analysis correctness proof, implementation and experimental results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–37, San Diego, CA, January 1998.

[8] Z. Budimlic and K. Kennedy. Optimizing java: Theory and practice. *Concurrency: Practice and Experience*, 9(6), June 1997.

[9] 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.

[10] A. Chien, J. Dolby, B. Ganguly, V. Karamcheti, and X. 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.

[11] A. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ – a C++ dialect for high-performance parallel computation. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*, pages 76–95, March 1996.

[12] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[13] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[14] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensative interprocedural computation of pointer-induced aliases and side effects. In *Twentieth Symposium on Principles of Programming Languages*, pages 232–245. ACM SIGPLAN, 1993.

[15] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[16] J. Dean, C. Chambers, and D. 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.

[17] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 230–241, 1994.

[18] A. Diwan, K. S.McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 106–117, Montreal, Canada, June 1998.

[19] J. 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.

[20] J. Dolby and A. A. Chien. An evaluation of automatic object inline allocation techniques. In *Proceedings of the Thirteenth Annual Conference on Object-Oriented Programming Languages, Systems and Applications (OOPSLA)*, Vancouver, British Columbia, October 1998. Available at `http://www-csag.cs.uiuc.edu/papers/oopsla-98.ps`.

[21] J. R. Ellis and D. L. Detlefs. Safe, efficient garbage collection for c++. Technical report, Xerox Palo Alto Research Center, June 1993.

[22] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[23] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proceedings of the Workshop for Languages and Compilers for Parallel Computing*, 1995.

[24] K. E. Gorlen, S. M. Orlow, and P. S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons, 1991.

[25] C. Hall, S. L. Peyton-Jones, and P. M. Sansom. *Functional Programming, Glasgow 1994*, chapter Unboxing Using Specialization. Workshops in Computing Science. Springer-Verlag, 1995.

[26] M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, 1991.

[27] G. Hamilton, editor. *JavaBeans 1.01 Specification*. Sun Microsystems, Mountain View, CA, 1997. published online at `http://www.javasoft.com/beans/docs/beans.101.ps`.

[28] U. Hölzle, C. Chambers, and D. 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.

[29] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336, June 1994.

[30] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 235–249, 1992.

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

[32] H. D. Pande and B. G. Ryder. Static type determination and aliasing in c++. Technical Report LCSR-TR-250, Laboratory of Computer Science Research, July 1995.

[33] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. 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.

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

[35] J. Plevyak and A. 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.

[36] J. Plevyak and A. 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.

[37] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[38] Z. Shao, J. H. Reppy, and A. W. Appel. Unrolling lists. In *ACM Conference on Lisp and Functional Programming*, June 1994.

[39] A. A. Stepanov and M. Lee. The standard template library. ISO programming language C++ project. Technical Report X3J16/94-0095, WG21/NO482, Hewlett-Packard, May 1994.

[40] D. Stoutamire and S. Omohundro. Sather 1.1, draft. Available online from `http://www.icsi.berkeley.edu/s̃ather/Sather-1.1.ps`, August 1995.

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

[42] M. Tofte and L. Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):734–767, July 1998.

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

[44] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1991.