Submittted for Publication

Automatic Interprocedural Optimization for Object-Oriented Languages

John Plevyak and Andrew A. Chien

Department of Computer Science
1304 W. Springfield Avenue
Urbana, IL 61801
(217) 244-7116
{*jplevyak,achien*}*@cs.uiuc.edu*

February 29, 1996

Abstract

The structure of object-oriented programs differs from that of procedural programs, requiring special compilation techniques to obtain efficiency. Object-orientation introduce additional layers of abstraction which separate the implementations of data structures and algorithms from their points of use. The result is smaller functions, increased data dependence of control flow, and an increase in potential aliasing. We present an automatic interprocedural optimization framework, which resolves these differences through interprocedural analysis and transformation; enabling a dynamically-typed pure object-oriented language to match the performance of C on a set of standard procedural benchmarks. For two standard object-oriented benchmarks we also compare this framework to the supplied annotations (e.g. `inline`, `virtual` and templates) for the hybrid OO-procedural language C++ [51], and show that the framework obtains up to 6 times the performance of the annotated C++ code.

1 Introduction

The structure of object-oriented (OO) programs differs from that of procedural programs, reflecting to a greater degree the abstractions of the problem rather than the implementation of the solution. This is a natural consequence of the OO programming style which encourages information hiding, the encapsulation of implementations behind abstract interfaces. Compilers for such languages must be able to reach beyond these abstractions, essentially breaking the encapsulation, to build efficient implementations. Program level abstractions are wrapped in a set of member functions which represents the interface of an object. These functions can be very small; for example, accessor functions abstract pieces of data, simply returning the value of a member variable. In order to remove the overhead of executing a function call each time an abstraction is used, the compiler must apply an interprocedural transformation; it must inline member functions.

The other significant property of object-oriented programs is an emphasis on code reuse. Abstractions are described by difference (inheritance) or by parameterization (templates [51] or generics [30]). The actual arguments to a function can be of any class which supports the operations used by the function.[1] This *polymorphism* enables a single function to operate on many types of data by using the abstraction. This presents two problems for the compiler. First, an argument may have different physical representations, requiring a generic function to manipulate it indirectly. Second, the compiler can no longer inline member functions since they may differ between the different actual classes.

[1] The type system may impose restrictions, requiring a common base class (declared type), template or *signature* [4].

We present an automatic interprocedural compilation framework based on interprocedural analysis and transformation which breaks through encapsulation and automatically builds specialized versions of polymorphic abstractions. Thus we can produce implementation of object-oriented programs using high level abstractions which are as efficient as their procedural equivalents. Moreover, the programmer is not required to alter the program expression in order to obtain this efficiency. For a set of standard procedural benchmarks, we show that a dynamically-typed pure object-oriented language (see Section 2.5) can match the efficiency of C [33]. In hybrid languages, like C++ [21], programmers can summarize interprocedural information and direct optimization through `inline` hints, careful placement of `virtual` declarations, and the use of derivation with templates [52]. For two standard object-oriented benchmarks, we show that the annotations provided by the programmer are not sufficient and that our automatic approach improves performance by up to 6 times over the C++ versions.

The remainder of this paper is organized as follows. In Section 2 we detail the characteristic features of object-oriented programs and their effect on efficiency. Section 3 describes our interprocedural optimization framework and Section 4 compares its performance with that of C and C++ for a set of standard benchmarks. Finally in Sections 5 and 6 we discuss the implication of these results and related work, and then conclude.

2 Object-Oriented Programs and Languages

This section examines the characteristics of object-oriented programs which a compilation framework must address. OO programs tend to have smaller functions, more data dependent control flow, and more potential aliases than procedural programs [27, 6]. These features can decrease performance, and their effects compound. The increased data dependence of control flow means that more of the function calls will require indirection (virtual function calls). The greater number of aliases means that more data will have to be spilled from registers for the larger number of function calls. Since these differences vary across OO languages, we examine two representatives of the ends of the static vs. dynamic spectrum. These are: hybrid languages (e.g. C++) which are built on top of a statically-typed procedural language, and dynamically-typed pure object-oriented languages. For each we discuss implementation implications. Finally, we introduce the particular dynamically-typed pure object-oriented language used to evaluate the framework.

2.1 Function Size

The small function size of object-oriented programs is a result of two programming techniques: encapsulation and programming by difference. These techniques are supported by methods (member functions) and inheritance respectively. Methods are used to describe the interface to the object, physically embodying the abstraction boundary. Using inheritance, the programmer partitions the program into functions representing a general solution and a set of variation points. These variation points are likewise delimited by function boundaries.

The effects of object-orientation on program characteristics have been confirmed empirically for C++ and C. In one study, Calder et al. [6] found that the instructions to invocation ratio for C++ was less than half that of C (48.7 vs. 152.8). Moreover, the basic block size for C++ was slightly smaller than that of C. In addition to the overhead of the function call itself, small function and basic block size make it harder for modern microprocessors to extract the instruction level parallelism they depend on for speed.

2.2 Data Dependent Control Flow

In object-oriented programs, references (pointers) are polymorphic; that is, they can point to objects of more than one class. Methods invoked on these objects are dependent on the actual class of the object. In general, the function executed is determined by a combination of the selector (virtual function name or pointer) and the actual class of the object, both of which may vary at run time. Thus, the control flow of the program is dependent on the data flow, neither of which can be resolved locally.

Data dependent control flow complicates inlining, increases the cost of function calls and decreases the precision of interprocedural analysis. Inlining, if it can be done at all, must be done conditionally, based on the class of the target object [10, 27, 5]. Function calls are more expensive since they require a set of

conditionals [28], or an indirect function call [52]. Finally, since data dependence causes interprocedural control flow to be ambiguous, analyses which depend on such information become less precise.

2.3 Aliasing

Since objects may be referenced by pointer, their instance (member) variables are potentially aliased. As a result, these variables generally cannot be cached in registers across function calls or assignments through pointers. Since member variables are implicitly scoped in C++ (they need not be accessed through the `this` pointer), this performance consequence is relatively opaque (i.e. may not be readily apparent to the programmer constructing or using the abstraction). Moreover, the potential alias problem is exacerbated by high function call frequency.

```
template<class V> class Array2D {
  int outer;
  int inner;
  V data[MAXDATA];
  V at(int theouter,int theinner);
}

template<class V> V Array2D<V>::at(int theouter,int theinner)
{
  return data[(inner * theouter) + theinner];
}

...
  for (j = 0; j < a->outer ; j++ )
    for (i = 0; i < a->inner ; i++ )
      b->compute( a->at(j,i) );
```

Figure 1: Example of member variable aliasing in a two-dimensional array accessed by linearization.

An example of this effect appears in Figure 1. The `Array2D` template encapsulates a contiguously allocated two dimensional array object. The function `at()` accesses an element of that array by the standard technique of linearizing the indices [20]. In isolation, this member function requires a memory access to retrieve `inner`. Inlined into the `for` loop below, the load of `inner` cannot be removed from the loop unless it can show that `inner` cannot be changed by `compute`. Moreover, if the class of `b` is not known precisely this determination is more difficult, since alias analysis must be conservative across all possible paths taken.

2.4 Hybrid OOPLs

Hybrid languages consist of object-oriented features atop an underlying base language. Consequentially, the object-oriented features are distinguished from procedural features (e.g. in C++ object are different from *primitive* types and virtual from non-virtual functions [51]). As a result, C++ preserves C data types and call mechanisms and can be implemented by locally mapping object-oriented features to the underlying procedural programming model. For example, the *messages* of Smalltalk [23] can be implemented as indirect calls through a *virtual function table* [21]. By annotating the program with information about where to use object-oriented features, the programmer provides interprocedural information to the compiler and directs optimization. For example, non-virtual member functions cannot be overridden, enabling them to be called without indirection. Similarly, `inline` indicates that a particular transformation would likely be profitable.

2.5 Dynamically-typed Pure OOPLs

Dynamically-typed pure object-oriented languages [23, 53] eliminate some of the distinctions made in hybrid languages. They do not distinguish between objects and primitive types or virtual and non-virtual functions. They have no type declarations, prototypes, *inline* hints or templates. They do not require or enable programmers to choose implementation mechanisms. However, such a high level approach has implementation consequences. Local translations, such as those undertaken for hybrid languages, are sufficient only to produce inefficient code. Differentiating primitive types and invoking dynamically bound functions at run time can be expensive, requiring more aggressive techniques [10, 27]. Finally, since every function call is nominally virtual and every reference nominally polymorphic, pure object-oriented languages represent a "worst case" for a compiler.

2.6 Basis of Comparison

We use a pure object-oriented languages as a basis for evaluating the effectiveness of the compilation framework. The benchmarks were translated from C and C++ into the sequential subset of Concurrent Aggregates (CA) [14, 13, 15]. CA is a simple object-oriented language resembling Smalltalk with Lisp syntax. In CA, classes simply list their instance variables and superclass; methods the class they are on, their arguments and an expression to evaluate; and expressions are either a message send or a basic control structure[2] (see Figure 1 for the syntax). All the built-in classes and operations are defined in a standard prologue in terms of *primitives* [23] and can be overridden. Moreover, in CA, like SELF [53], accessor methods must be used to manipulate instance variables. Thus, a naive translation of CA would be extremely inefficient, with a ratio of useful instructions to dynamically bound message sends (indirect function calls) of less than one.

class	(`class` name (superclass) instance_var*)
method	(`method` class_name selector (parameter*) expression)
message	(selector object argument*)
control	(`if` expression expression expression) (`while` expression expression*)
primitives	(`primitive` name expression*)

Table 1: Concurrent Aggregates Syntax

3 The Compilation Framework

To produce efficient code for object-oriented languages, we determine which program properties are static, and then transform the program taking advantage of this information. Since transformation changes the program, additional properties may become statically known, enabling additional optimization. Instead of iteratively transforming and reanalyzing, we apply a flow analysis technique which captures the effects of optimization to enable optimization decisions to be based on the range of potential transformations. The phases of compilation are:

- Analysis - Context sensitive interprocedural flow analysis
- Cloning - Selection of classes and functions to be specialized
- Specialization - Creation of class and functions with specialized information
- Optimization - Enabled standard optimizations

[2] These structures are not strictly necessary; see Section 5.1 for a discussion of *blocks* [23].

The analysis phase constructs an interprocedural call graph which is used for other analyses and general interprocedural optimizations (e.g. constant propagation). The cloning phase uses static estimation [54] and a set of optimization criteria to determine which specific instances of general polymorphic classes and function to create. The specialization phase optimizes classes by constructing new dispatch tables, unboxing and specializing functions by statically binding and 'if conversion' [7, 28, 6] of virtual function calls. The optimization phase inlines functions, use aliasing information to promote member and global variables to locals, determines array aliasing, and applies such traditional transformations as invariant lifting, strength reduction, constant folding and global common subexpression elimination.

3.1 Example

We use polymorphic array multiplication as a running example to illustrate the optimization framework because it is well known and simple. For the same reasons we use C++ syntax but access instance variables via methods and omit type declarations and other annotations (e.g. `virtual`). The code in Figure 2 declares the two-dimensional polymorphic array class `Array2D` and the `innerproduct` and matrix multiply `mm` functions. These functions are then used to multiply two arrays containing integers `ai` and `bi` into an integer array result `ri`, and two arrays containing floats `af` and `bf` into a float array result `rf`. Except for syntax, this is essentially the same as the CA code for the benchmark in Section 4.

```
class Array2D : Array {
  inner;
  outer;
  at(theouter, theinner);
  at_put(theouter, theinner, value);
  innerproduct(a,b,row,column);
  mm(a,b);
};

Array2D::at(theouter, theinner) {
  return (*this)[(inner() * theouter)
               + theinner];
}

Array2D::at_put(theouter, theinner, value) {
  (*this)[(inner() * theouter) + theinner]
    = value;
}
```

```
Array2D::innerproduct(a,b,row,column) {
  result = a.at(row,0) + b.at(0,column);
  for (i=1;i<a.inner();i++)
    result += a.at(row,i) + b.at(i,column);
  at_put(row,column,result);
}

Array2D::mm(a,b) {
  for (i=0;i<b.outer();i++)
    for (j=0;j<a.inner();j++)
      innerproduct(a,b,i,j)
}

main() {
 Array2D ai,bi,ri;  // L1
 Array2D af,bf,rf;  // L2
 ...
 ri.mm(ai,bi);      // L3
 rf.mm(af,bf);      // L4
}
```

Figure 2: Polymorphic matrix multiplication example.

3.2 Analysis

During the analysis phase, functions and classes are parameterized much like templates with the classes and function pointers which their variables may be at runtime. These parameterizing classes and functions may themselves be parameterized, enabling deep and/or recursive call paths or data structures to be analyzed. The technique used is iterative context sensitive flow analysis [41], which operates by constructing an approximation of the interprocedural data flow graph by abstract interpretation of function dispatch [39], and iteratively extending the context sensitivity.

The result of analysis is context sensitive information where a context is given in terms of call paths for functions and creation points for objects. More precisely a context c is a function f, called from a statement

s in context c' on an object created at statement s' in context c''.[3] For each context the analysis provides the classes which each variable might point to, and likewise for the object creation points, the classes of member variables of objects created there. Furthermore, the analysis provides an interprocedural call graph over the contexts. The cloning phase uses this information to decide which contexts should be instantiated as unique functions and which sets of objects should be instantiated as unique classes. Essentially, the analysis phase treats the user's functions and classes as a set of templates and determines how they might be instantiated.

For our example, the result of analysis is that the objects created at line `L1` contained integers (let us call them **Array2D(int)**), and those created at `L2` contained floats (**Array2D(float)**). It also shows that `mm()` was called from `L3` on three objects of **Array2D(int)** and from `L4` on objects of **Array2D(float)**. Furthermore, it shows that within these two versions of `mm()` the corresponding versions of `innerproduct()` were called and within them the corresponding versions of `at()` and `at_put`. Thus, the `at()` within `innerproduct()` on **Array2D(int)** is known to return an integer.

3.3 Cloning

In the cloning phase, the compiler determines which functions and classes to specialize. Based on static estimation of the frequency with which functions and classes are used, the compiler determines which regions of the program are performance critical. The contexts which provide the most precise information about those regions are selected for specialization based on optimization criteria such as static binding of function calls and unboxing of local and member variables. In order to ensure that the selected functions and classes can be created and specialized, other functions and classes may have to be cloned. For example, in Figure 3, a specialization of `innerproduct()` induces a specialization of `mm()`.

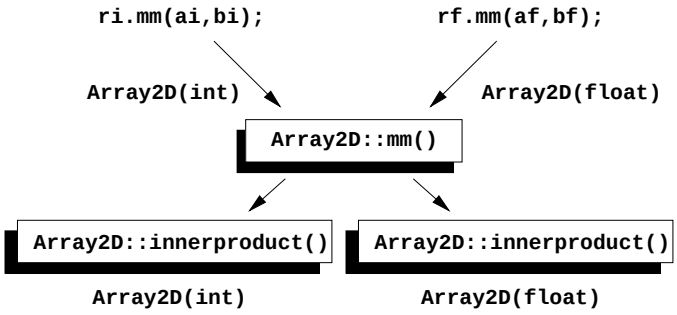

Figure 3: Example of how specialization of `innerproduct()` induces specialization of `mm()`

An iterative algorithm [42] is used to determine which additional functions and classes should be specialized. Function specialization induces specialization of the caller when the calling context is required to determine which specialized functions to call. Likewise, function specialization induces class specialization when the class identifier is needed at a call site to determine which specialized function to call. Finally, the specialization of classes induces function specialization based on the requirement that objects of the specialized class must be created at unique points in the cloned program (so that they can be constructed with the appropriate structure and class identifier).

While in Figure 3 a virtual function call can be substituted for a specialization, in general the function could be specialized based on the classes of any argument, or even class of a member variable of an argument. When a function has been so specialized, and is called from a context which requires dynamic dispatch, the virtual function index [52] and target object class no longer uniquely determine the function. We modify the dispatch mechanism to include a statement identifier (which can be folded into the virtual function index) to enable the correct function to be determined.

[3] The recursion in the definition is headed off by having `main` called from a distinguished context c_0 on an object created at s_0 in c_0.

3.4 Specialization

Once the functions and classes are cloned, and the resulting methods are specialized with respect to the contexts which they represent. The memory map of a class clone is specialized with respect to the unboxed types of its member variables. Likewise, the dispatch table for the class clone is modified to represent the functions specialized to operate on the class clone. Functions are specialized with respect to the types of their arguments and local variables, and wrapper functions are constructed which map between boxed and unboxed representations for arguments when a dynamic dispatch is required to functions with incompatible specialized arguments [47, 26, 36].

The interprocedural call graph is then updated with respect to the realized call graph. Since it was generated during analysis over contexts, the edges for a function are simply those of all its contexts. When there is only one edge from a call, that call is statically bound. When there is more than one edge and the number of edges is less than a constant (in our implementation five), the call site is 'if converted'. That is, the dispatch is converted into a set of conditioned statically bound calls where the conditions test either the class of the target object or the virtual function index of a virtual function pointer [9]. In later phases, these statically bound calls can be inlined.

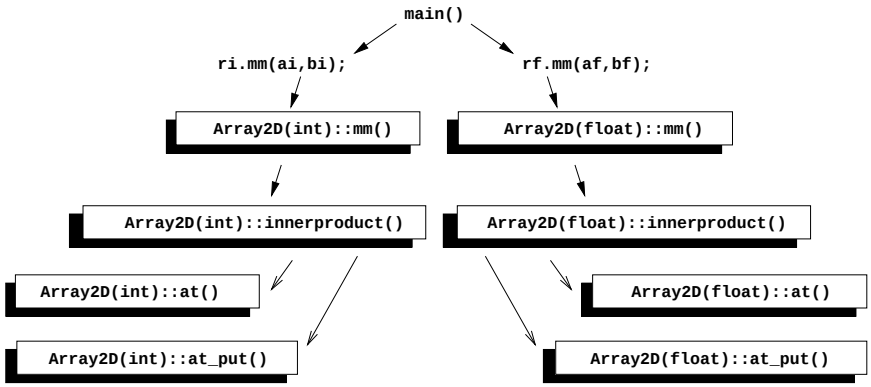

Figure 4: Example of specialization of polymorphic matrix multiply.

For our example, the specialized classes **Array2D(int)** and **Array2D(float)** are created with the knowledge of the types of `inner`, `outer` and the array elements. The constructors at `L1` and `L2` are specialized to create objects of these new classes. The access functions `inner()` and `outer()` are specialized to extract unboxed integers. Likewise the specialized versions of `at()`, `at_put()`, `innerproduct()` and `mm()` are created. Finally, the call graph is updated so that, for instance, `innerproduct` on **Array2D(float)** calls `at_put()` on **Array2D(float)** as in Figure 4.

3.5 Optimization

The specialized methods and classes produced by cloning are similar to template instantiations in that the information they contain has been made more precise by code replication. However, function call density is still very high and crossing function boundaries for each of the many small functions would be expensive. Also, the large number of pointer based member variable accesses would prevent effective register use. Even when these problems have been addressed by inlining and promotion of member variables to locals, the residue of high level abstractions can leave the code containing many redundancies. These are addressed with standard low level optimizations.

3.5.1 Inlining

We inline based on heuristics using a combination of static estimation [54] and size constraints to remove the cost of crossing procedure boundaries. The effect of *splitting* [8] which preserves the information obtained by runtime type or function pointer checks within functions, can be obtained by merging checks with identical

conditions [44]. Since many small virtual function call bodies contain only a few instructions, and since the interprocedural call graph can be used to determine when a function is not called by anyone and eliminate it, inlining need not result in a large code size increase [42]. For example, both versions of the `innerproduct()` function will be inlined into their corresponding `mm()` call sites and the functions will be eliminated since they have no other callers.

3.5.2 Variable Promotion

Since member variables are ubiquitous, accessed by reference and potentially aliased (see Section 2.3), they represent a large potential overhead. Using the interprocedural call graph and the object creation context information provided by the analysis, we estimate whether a function call or member variable access might alias a given member variable. Since pure languages do not allow pointers into objects, only other accesses to the same member variable of objects created at the same point as the member variable in question can alias it. The interprocedural call graph enables us to approximate the statements reached by a function call. Member variables unaliased over a range of statements are promoted to locals and can be allocated to registers.

Likewise, global variables can be promoted to local variables. Since global variables are uniquely named and cannot be pointed to, their aliasing pattern can be easily determined from the interprocedural call graph.

```
Array2D::mm(a,b) {
  for (i=0;i<b.outer();i++)
    for (j=0;j<a.inner();j++)
      innerproduct(a,b,i,j)
}
```

```
Array2D::mm(a,b) {
  tmp_outer = b.outer();
  tmp_inner = a.inner();
  for (i=0;i<tmp_outer;i++)
    for (j=0;j<tmp_inner;j++)
      innerproduct(a,b,i,j)
}
```

Figure 5: Example of promotion of member variables `inner` and `outer` to locals.

In our example, the `inner` and `outer` member variables are part of the `Array2D` object which is potentially aliased. However, using the call graph we can determine that for the objects created at `L1` and `L2` these member variables are not changed within any function called from `mm()`. Thus, as in Figure 5 we can promote the member variables to local temporaries `tmp_inner` and `tmp_outer` and hoist them out of the loop.

3.5.3 Array Aliasing

In the same way that alias information can enable promotion of member variables, it can approximate aliasing for arrays. Since pure languages does not allow pointers into the middle of arrays, absolute and/or symbolic analysis can be used to determine that array accesses do not conflict. We use a simple creation point test to estimate interprocedural array aliasing and combine it with simple symbolic analysis to enable array references to be lifted and common subexpression eliminated.

```
if (a.at(i) > a.at(i+1)) {
  tmp = a.at(i);
  a.at_put(i,a.at(i+1));
  a.at_put(i+1,tmp);
}
```

```
tmp_i = a.at(i);
tmp_i1 = a.at(i+1);
if (tmp_i > tmp_i1) {
  a.at_put(i,tmp_i1);
  a.at_put(i+1,tmp_i);
}
```

Figure 6: Example of common subexpression elimination of array operations from bubble sort.

For example, in Figure 6 from the bubble sort benchmark (see Section 4), the inner loop contains two array reads in the conditional and two in the body (left). We can determine by simple analysis over the call tree that the array could not be written between the first call to `a.at(i)` and the second. Thus, we can lift and common subexpression eliminate the `a.at(i)` in the loop (right).

3.5.4 Other Traditional Optimizations

Since the abstractions of object-oriented programming are often used to hide the representation of data, ostensibly simple operations may require many instructions. For example, the CA language does not support native multi-dimensional arrays. These are constructed out of single dimension arrays with member variables containing the dimension sizes and linearization functions (as in Figure 2). When multi-dimension arrays are used in loops, the member variables can be promoted and the linearization operations strength reduced and moved outside the loop. With these optimizations, matrix multiply of multi-dimensional arrays in CA is as fast as C (see Section 4), even though the array operations are abstracted and ostensibly require much more work.

3.6 Context

This optimization framework is embodied in the Concert [12] retargetable compiler for concurrent object-oriented languages which currently supports both Concurrent Aggregates (CA) and ICC++ (a parallel dialect of C++). In this paper we are concerned only with the sequential subset of CA. Readers interested in transformations specific to concurrent languages and parallel implementations are directed toward [44, 43].

4 Results

We use a standard benchmark suite to evaluate and compare the performance of CA, C and C++. The Stanford Integer Benchmarks, Richards and Delta Blue were used to evaluate the SELF language by Chambers [10] and later by Hölzle [27]. The Stanford Integer Benchmarks are small procedural codes. The CA versions use encapsulated objects for the primary data structures, but otherwise follow the C code structure. Richards and Delta Blue both use polymorphism, some of which can be removed at compile time by templates or cloning (i.e. parametric polymorphism) and some which cannot (the task queue and constraint network respectively). The CA versions of these codes follow the C++ encapsulation and code structures.

4.1 Benchmarks

The Stanford Integer Benchmarks consist of bubble sort (**bubble**), integer matrix multiply (**intmm**), a permutation generator (**perm**), a 15-puzzle solver (**puzzle**), the N-queens problem (**queens**), the sieve of Erastothenes (**sieve**), the towers of Hanoi (**towers**), and a program to construct a random binary tree (**tree**). Richards is an operating system simulator which creates a number of different tasks which are stored in a queue and periodically executed. Delta Blue [46] is a constraint solver which builds a network, solves it a number of times and removes the constraints. Two different test cases are provided for Delta Blue, **Chain** which builds a chain of `Equal` constraints, and **Projection** which builds two sets of variables related by `ScaleOffset` constraints. The C++ codes are annotated by declaring functions `virtual` only when necessary [27], including inline accessors, and, in Delta Blue, by the use of a `List` template.

4.2 Methodology

We compare the performance of CA codes translated from the original sources.[4] Our compiler uses the GNU compiler [49] as a back end, enabled us to control for instruction selection and scheduling differences by using the same version (2.7.1) for both the back end of our compiler and the C and C++ benchmarks. All tests were conducted on an unloaded 75Mhz SPARCStation-20 with Supercache running Solaris 2.4. We present both individual results and summarized results for the procedural and object-oriented benchmarks. The individual results are the average execution time of 10 repetitions of each benchmark at each optimization setting normalized to the execution time of C/C++ at -O2. The summarized results are the geometric means of the normalized times over all the benchmarks at each optimization setting. This effectively constructs a synthetic workload in which each benchmark runs for the same amount of time for C/C++ at -O2.

[4] Our thanks to Craig Chambers and Urs Hölzle for making the codes available.

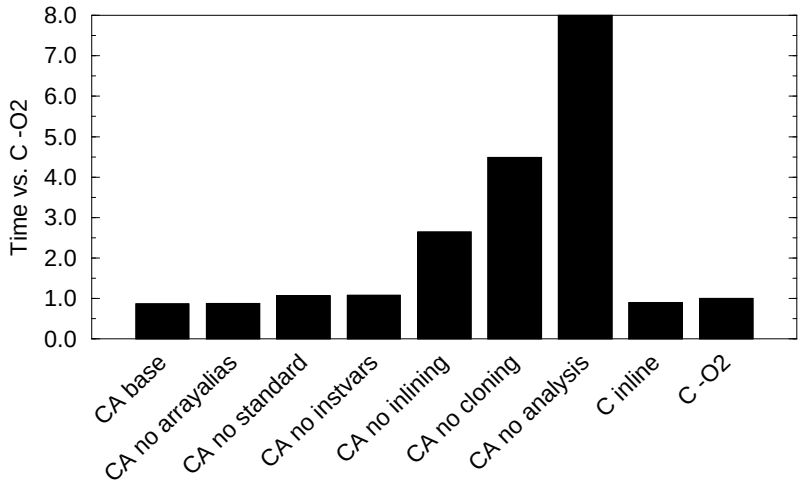

Figure 7: Geometric mean of execution time relative to **C -O2** for the Stanford Integer Benchmarks vs. optimization settings.

4.3 Procedural Codes: Overall Results

Figure 7 graphically summarizes the execution time of the Stanford Integer Benchmarks under various optimization settings relative to the C optimized at -O2. Overall, the results show that at full optimization the performance of the dynamically-type pure OO codes matches that of C. The different bars report the cumulative effect of disabling optimizations. In order to make a comparison with C++ easier, the **no inlining** bar does not prevent inlining of accessors or operations on primitive data types (e.g. integer add) and the **analysis** bar only disables flow sensitivity.

Each optimization contributes to the overall performance which would otherwise be an order of magnitude (9.2 times) less than C. Context sensitive flow analysis alone provides a factor of two by allowing more primitive operations to be inlined, and function calls to be statically bound. Cloning contributes another factor of two for essentially the same reasons, by making monomorphic versions of polymorphic code. Inlining operates on statically bound calls, more than doubling performance by eliminating call overhead. Instance variable promotion enables many of the **standard** optimizations which, together with array alias analysis provide the last twenty percent of overall performance.

These results demonstrate that for such procedural kernels, the cost of the unused flexibility of OO features can be eliminated. The remaining differences in performance reflect the low level optimizations favored by the GCC compiler and the code structure more than any inherent language advantage. For example, GCC strength reduces array accesses based on `sizeof(int)` using a simple heuristic which is easily confused by the RTL-like output of our compiler. Likewise, computing booleans into intermediates can inhibit direct use of condition codes. On the other hand, GCC only inlines functions which appear previously in the same file, and the standard `malloc` routine is relatively inefficient.

4.4 Procedural Codes: Individual Results

Figure 8 reports the individual performance of the benchmarks. Overall, the results are split, with **CA base** outperforming **C -O2** in some cases and **C -O2** outperforming **CA base** in two. **C inline** increases that number by one, adding **towers** (a highly recursive code) to the list for which C is faster. Since these codes are largely monomorphic they can be analyzed easily, and they differ only in their control structures and function boundaries. The CA codes are faster because of relatively more aggressive inlining except **trees** which allocates many objects. Even though the CA code garbage collects eleven times during each run, it is still more efficient than `malloc()`. On the other hand, CA does not support `break`, and the resulting additional condition in `while` loops drops CA performance on **puzzle** by almost a factor of two. Discounting these special cases, individual benchmark performance is nearly identical.

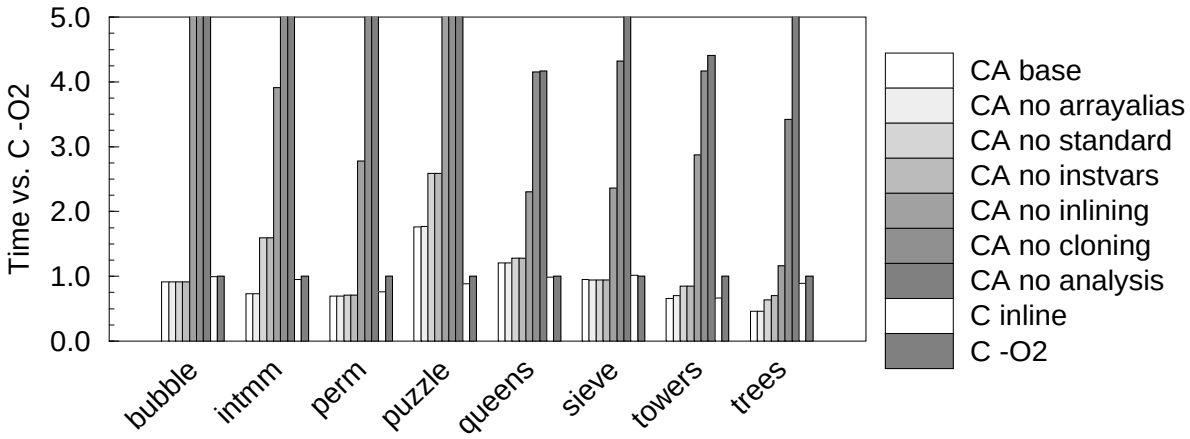

Figure 8: Performance relative to **C -O2** on the Stanford Integer Benchmarks

Different optimizations had larger effect on different benchmarks, indicating their individual importance. The **bubble** sort and permutation (**perm**) programs are heavily dependent on inlining (for the swap) which provides most of the performance. The **trees** and **puzzle** programs benefit directly from instance variable promotion, in **trees** case because of the heavy use of the `left` and `right` child instance variables. Matrix multiply (**intmm**) and **puzzle** are loop based, and depend on the standard optimizations, and in particular strength reduction which is enabled by instance variable promotion (for the inner loop dimension). Finally, **towers** and **puzzle** benefit from array alias analysis because the code repeatedly accesses the elements at the same array offsets.

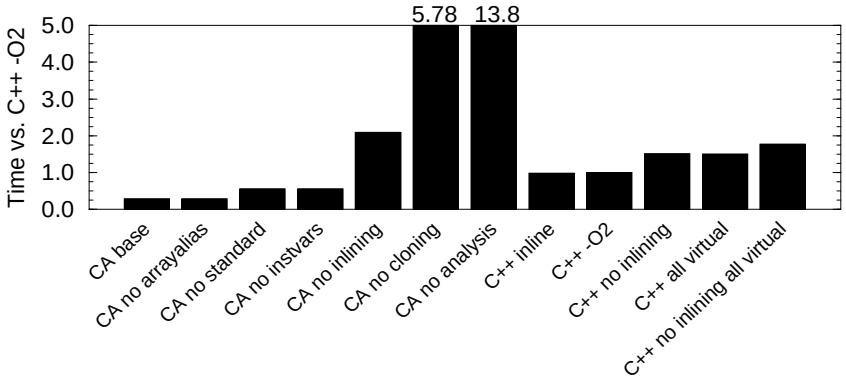

Figure 9: Geometric mean of execution time relative to **C++ -O2** for the OOP Benchmarks vs. optimization settings.

4.5 Object-Oriented Codes: Overall Results

Figure 9 graphically summarizes the execution time for the OO benchmarks for CA at seven optimization settings and for C++ versions at five. Overall, the CA compiler produces much better performance, a factor of four improvement, over the C++ compiler even with user provided annotations and automatic inlining enabled (-O3). Again, the individual optimizations made their contributions starting from a initial performance point an approximately order of magnitude (13.8 times) worse than C++. Context sensitive flow analysis provided a factor of 2.4. This is more than the factor of two for the procedural codes, indicating its relative importance for OO codes. Likewise, cloning was responsible for approximately a factor of 2.5. Inlining contributed a factor of four, again showing its relative importance for OO codes. Finally, standard low level optimizations enabled by instance variable promotion contributed a factor of two.

We evaluated the performance of the benchmarks with the C++ compiler at five optimization settings, including the base (-O2), automatic inlining (-O3), without any inlining, with all virtual functions, and without any inlining and all virtual functions. Automatic inlining improved performance by approximately two percent, indicating that most of the automatically inlinable functions had been annotated. Disabling either inlining or static binding annotations reduced performance by 50 percent, and with both were disabled, performance dropped by 70 percent. Performance of the CA code without inlining was comparable to that of the C++ compiler without inlining. However, as we will see in the next section, the individual results vary, indicating this is just coincidental.

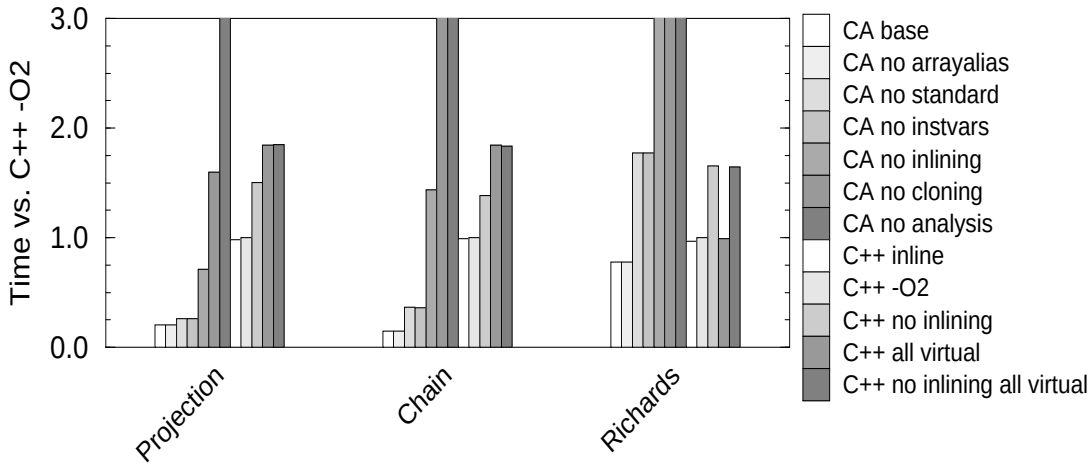

Figure 10: Performance of CA and C++ relative to **C++ -O2** on OOP Benchmarks

4.6 Object-Oriented Codes: Individual Results

Figure 10 reports the results for the Richards, Delta Blue chain and projection individual benchmarks under various optimization settings of both the CA compiler and the C++ compiler. The CA versions vary from almost six times faster (**Chain**) to twenty-five percent faster (**Richards**) than **C++ -O2**. In the case of Delta Blue, this difference is attributable to many calls to small functions. For example, in object-oriented fashion, Delta Blue uses a general `List` container object with a member function which applies a function pointer (or selector [23]) across its elements (e.g. `do:` in Smalltalk or `map` in Scheme). The CA compiler clones and inlines both call sites, turning this into a simple C style loop containing operations directly on the elements, while the C++ compiler does not. For Richards, the performance difference is primarily a result of optimizations enabled by instance variable promotion. Richards uses an object to encapsulate its current state including the head of the task queue, and manipulation of this state makes up the largest part of the execution time.

The different C++ optimization settings produced different results for the different benchmarks, much as they did for CA. For **Richards**, disabling inlining decreased performance by 65 percent. However, making all methods `virtual` has no effect on performance at all. This is because **Richards** is largely a procedural code where the central switch statement has been replaced with a virtual function call (the `run` method on `Task` objects). On the other hand, Delta Blue uses accessor functions and other small methods for encapsulation of object state. For **Chain**, disabling inlining decreases performance by 40 percent and for **Projection** the impact is a 50 percent decrease. Furthermore, since Delta Blue does most computation through methods, making all methods `virtual` (which effectively prevents inlining of methods as well) reduces performance by 85 percent, and the performance is unchanged when inlining is disabled as well.

These results show that for these benchmarks, the overhead from object-orientation can be removed automatically. Furthermore, it they show that the annotations provided by the C++ programmer are not sufficient alone to optimize the programs.

5 Discussion and Related Work

Interprocedural optimization has been growing in popularity, especially with the recognition of the contribution of compilers to the SPECmarks (and hence the recognized performance) of commodity microprocessors. The necessity for interprocedural optimization for object-oriented programs comes as no surprise. The assumption of interprocedural optimization as a part of C++ implementations is embodied in the `inline` hints, and the implicit inline hints associated with declaring code in class definitions. Likewise, the template repository [52], used to minimize the number of redundant instantiations of templates automatically, depends on global information. Many analyses, and in particular alias and type analysis, require information about the whole program [40, 55, 45], however summarization techniques exist [3] which can enable separate compilation. C++ addresses this problem by summarizing and importing parts of the program in header files and by compiling specialized versions of classes and functions (template instantiations) at link time [52]. Thus, fully automatic interprocedural optimization is the logical extension of current C++ practice.

5.1 Object-Oriented Language Features

In this paper, we focus on the additional complexity that object-orientation introduces. In particular, we chose to concentrate on the most uniquely object-oriented feature: virtual functions (also called late binding or dynamic dispatch) on polymorphic variables. Other features common in pure object-oriented languages include variations on generic arithmetic, run time error checking and *blocks* [23, 53]. However, these features are not unique to object-orientation. For example, in Smalltalk and SELF if an addition of two small integers causes an overflow, the result is a BigNum. Such systems are part of Scheme [16] and Common Lisp [32]. Likewise, array bounds and null pointer checks are part of the Pascal/Modula family. Finally, all control structures could be encoded using the blocks of Smalltalk and SELF, which are essentially the anonymous closures of Lisp and other functional languages.

5.2 Related Work

Many researchers have applied interprocedural flow sensitive analysis to determine type and/or class information from Shivers [48] through Stefanescu and Zhoi [50] and Jagannathan and Weeks [31]. In terms of object-oriented languages, the constraint based approach [39, 38] has been most recently extended to greater degrees of flow sensitivity by Agesen [1, 2] using the Cartesian product of the classes of function arguments. In contrast, our analysis [41] is flow sensitive with respect to both the classes of function arguments and the classes of member variables of data structures.

Cloning and specialization have been studied in the context of FORTRAN by Cooper [17] and Hall [24, 25]. In the context of object-oriented programs, customization [7] by Chambers and Ungar and later specialization [18] by Dean, Chambers and Grove have addressed the problem of selecting versions of functions to duplicate based on the classes of the functions arguments. Their decisions have been based on the classes of target objects, class hierarchy analysis [19] and/or profiling information. Our cloning and specialization technique is based on global flow analysis and includes specialization of data structures (classes). We discussed its use in the removal of dynamic dispatch in [42].

Using a simple analysis, which determines that a virtual function is never redefined, virtual functions can be statically bound at link time [5, 22]. However, inlining and optimization at link time requires decompilation techniques [35, 37]. Moreover, some transformations are difficult or impossible at this level. For instance, the semantic connection has been lost between the (link time constant) size and offsets within a data structure and the functions which operate on it, preventing high level data structure transformations like inlining of objects which the C++ programmer can achieve through templates.

Interprocedural alias analysis [55, 45] can determine when a member variable may be accessed through a pointer or in a called function. For C++, analysis that can determine the types of pointers [40] has been undertaken based on pointer alias analysis technology [34]. The resulting information can be used to statically bind virtual functions. The interprocedural call graph produced by our flow analysis should enhance these results. Likewise, Calder and Grunwald [5] also showed that relatively simple analysis can be used to statically bind many virtual functions. They propose using runtime checks (if conversion) to select

the appropriate virtual function at the call site and even inlining, but do not present implementation results. Moreover, the application of these techniques to cloning or specialization as yet to be explored.

This work most closely resembles that done for the SELF language [53]. While we obtained better raw speed, the SELF language and its integrated program development environment impose substantial restrictions which make direct comparison extremely difficult. SELF bound-checks arrays, produces BigNums when small integer operations overflow, and provides full source level debugging of optimized code. Nevertheless, our inlining system in similar to [7, 8, 10, 29, 27] in the use of class information and specialized function versions. However, their speculative inlining information is derived from the local prediction and preserving of information, polymorphic inline caches [28], or profiling.

6 Summary and Future Work

Object-oriented programs differ from procedural ones, requiring interprocedural optimization for efficiency. The compiler must be able to break through the layers of abstraction in the program specification to build an efficient implementation. We have presented an automatic interprocedural optimization framework sufficient to make object-oriented programs efficient with respect to their procedural analogues, and demonstrated them on a suite of standard benchmarks. We have shown that this framework applied to a dynamically-typed pure object-oriented language can improve performance up to 6 times over C++. This framework is implemented in and our experiments were conducted using the Concert [12] compiler to which we have recently added a front end for a C++ dialect (ICC++ [11]). We intend to use this platform to continue the evaluation of automatic interprocedural optimization.

7 Acknowledgments

We thank Vijay Karamcheti, Julian Dolby, Xingbin Zhang and the other members of the Concert project for their work on the Concert System.

The research described in this paper was supported in part by NSF grants CCR-92-09336, MIP-92-23732 and CDA-94-01124, ONR grants N00014-92-J-1961 and N00014-93-1-1086, NASA grant NAG 1-613, and a special-purpose grant from the AT&T Foundation. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809.

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] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of ECOOP '95*, pages 2–26. Springer-Verlag Lecture Notes in Computer Science No. 952, 1995.

[3] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Twenty First Symposium on Principles of Programming Languages*, pages 151–162, Portland, Oregon, January 1994.

[4] Gerald Baumgarter and Vince F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software – Practice and Experience*, 25(8):863–889, August 1995.

[5] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Twenty-first Symposium on Principles of Programming Languages*, pages 397–408. ACM SIGPLAN, 1994.

[6] 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.

[7] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–60, 1989.

[8] 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.

[9] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *OOPSLA '89 Conference Proceedings*, pages 49–70, July 1989.

[10] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Stanford, CA, March 1992.

[11] A. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ – a C++ dialect for high performance parallel computing. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*. Springer-Verlag, LNCS 742, 1996.

[12] Andrew Chien, Vijay Karamcheti, and John Plevyak. The Concert system – compiler and runtime support for efficient fine-grained concurrent object-oriented programs. Technical Report UIUCDCS-R-93-1815, Department of Computer Science, University of Illinois, Urbana, Illinois, June 1993.

[13] Andrew A. Chien. Concurrent aggregates: Using multiple-access data abstractions to manage complexity in concurrent programs. In *In Proceedings of the Workshop on Object-Based Concurrent Programming at OOPSLA '90.*, 1990. Appeared in OOPS Messenger, Volume 2, Number 2, April 1991.

[14] Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press, Cambridge, MA, 1993.

[15] Andrew A. Chien, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. Concurrent Aggregates language report 2.0. Available via anonymous ftp from cs.uiuc.edu in /pub/csag or from http://www-csag.cs.uiuc.edu/, September 1993.

[16] W. Clinger and J. Rees (editors). *Revised*^{4} report on the algorithmic lanuguage scheme. *ACM Lisp Pointers IV*, July-September 1991.

[17] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the R^n environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[18] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 93–102, 1995.

[19] Jeffrey Dean, Dave Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. Technical Report TR 94-12-01, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, December 1994.

[20] Jack J. Dongarra, Roldan Pozo, and David W. Walker. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of Supercomputing'93*, pages 162–171, 1993.

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

[22] Mary F. Fernández. Simple and effective link-time optimization of modula-3 programs. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–115, June 1995.

[23] Adele Goldberg and David Robson. *Smalltalk-80: The language and its implementation*. Addison-Wesley, 1985.

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

[25] Mary W. Hall, John M. Mellor-Crummey, Alan Clarle, and René G. Rodríguez. FIAT: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Workshop for Languages and Compilers for Parallel Machines*, pages 522–545, August 1993.

[26] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-second Annual ACM Symposium on Principles of Programming Languages*, pages 130–141. ACM SIGPLAN, 1995.

[27] Urs Hölzle. *Adaptive Optimization for* SELF*: Reconciling High Performance with Exporatory Programming*. PhD thesis, Stanford University, Stanford, CA, August 1994.

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

[29] 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.

[30] International Organization for Standardization. *Ada 95 Reference Manual*, version 6.0 edition, Dec 1994.

[31] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Twenty-second Symposium on Principles of Programming Languages*, pages 393–407. ACM SIGPLAN, 1995.

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

[33] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[34] 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.

[35] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[36] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proc. 19th symp. Principles of Programming Languages*, pages 177–188. ACM press, 1992.

[37] Digital Western Research Laboratory Om Project. http://www.research.digital.com/wrl/projects/om/om.html, October 1995.

[38] N. Oxhøj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *Proceedings of OOPSLA '92*, 1992.

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

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

[41] 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.

[42] 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.

[43] John Plevyak, Vijay Karamcheti, Xingbin Zhang, and Andrew Chien. A hybrid execution model for fine-grained languages on distributed memory multicomputers. In *Proceedings of Supercomputing'95*, 1995.

[44] John Plevyak, Xingbin Zhang, and Andrew A. Chien. Obtaining sequential efficiency in concurrent object-oriented programs. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, pages 311–321, January 1995.

[45] Erik 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.

[46] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Software – Practice and Experience*, 23(5):529–566, May 1993.

[47] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ML. In *SIGPLAN Conference on Programming Language Design and Implement ation*, pages 116–129, 1995.

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

[49] Richard Stallman. *The GNU C Compiler*. Free Software Foundation, 1991.

[50] Dan Stefanescu and Yuli Zhou. An equational framework for the flow analysis of higher-order functional programs. In *Proceedings of ACM Conference on Lisp and Functional Programming*, pages 318–327, 1994.

[51] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, second edition, 1991.

[52] Bjarne Stroustrup. *The Design and Evolution of C++*. Addsion-Wesley, 1994.

[53] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *Proceedings of OOPSLA '87*, pages 227–41. ACM SIGPLAN, ACM Press, 1987.

[54] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida USA, June 1994.

[55] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.