Type Directed Cloning for Object-Oriented Programs

John Plevyak and Andrew A. Chien

University of Illinois at Urbana-Champaign

Abstract. Object-oriented programming encourages the use of small functions, dynamic dispatch (virtual functions), and inheritance for code reuse. As a result, such programs typically suffer from inferior performance. The problem is that *polymorphic* functions do not know the exact types of the data they operate on, and hence must use indirection to operate on them. However, most polymorphism is parametric (e.g. templates in C++) which is amenable to elimination through code replication. We present a cloning algorithm which eliminates parametric polymorphism while minimizing code duplication. The effectiveness of this algorithm is demonstrated on a number of concurrent object-oriented programs. Finally, since functions and data structures can be parameterized over properties other than type, this algorithm is applicable to general forward data flow problems.

1 Introduction

Object-oriented (OOP) and concurrent object-oriented (COOP) programming languages have gained popularity because they provide programmers with useful tools for organizing programs. However, object-oriented programming techniques change the structure of programs significantly, typically incurring a performance degradation as a result. The reasons are fundamental to the programming models and include: encouraging programmers to use small functions, express new functionality by derivation from previous solutions (inheritance), share code (dynamic dispatch), and to separate use of operations from their implementation (data abstraction). Together, these techniques result in programs with high function call frequencies, and data dependent control flow.

These characteristics of object-oriented programs can result in poor performance on modern computers with high degrees of parallelism. Modern microprocessors rely on effective use of registers and instruction scheduling to achieve good performance. Object-oriented programs make frequent calls which, in addition to their own cost, disrupt instruction scheduling and register usage. To make matters worse, such programs allow the target of function calls to be data dependent, making inlining difficult or impossible[1] and compicating parallelization and concurrency optimization [30].

The key to eliminating the overhead of dynamic calls is *concrete type information*, knowledge of the implementation types that actually occur at function call sites. Such information can be obtained through global flow analysis [27, 26, 1, 28, 25] (across function boundaries and even across compilation units). These algorithms infer flow sensitive parameterizations for functions and data in the form of concrete type information. This information describes the pattern of

[1] Run time approaches are described in Section 5.

reuse of general (polymorphic) code for particular (monomorphic) situations. For example, a `Set` class might be able to contain any type of object, but a particular instance of `Set` might contain only `Circle` objects. The code operating on such instances could be optimized for the type of contents. Unfortunately, flow analysis results cannot be used directly for cloning, because the natural candidates for replication, contours [31], are too numerous and because standard dispatch mechanisms cannot select between them at runtime.

We present a cloning algorithm which minimizes the number of clones by replicating functions based on optimization criteria such as minimization of dynamic dispatch, unboxing opportunities and data layout. This is coupled with a call site specific dispatch mechanism to enable the selection of the appropriate clone by any remaining dynamic dispatches. We illustrate the efficiency and effectiveness of this algorithm through application to a suite of programs. Its efficiency is reflected in the modest code size increases (a range from -20% to +70%). The effectiveness is demonstrated by the elmination of dynamic dispatches resulting from parametric polymorphism in these programs. In our suite of pure concurrent object-oriented programs this results in static binding of approximately 99% of all calls and, through inlining, elimination of 45% to 99% of these calls. Thus, cloning reduces the number of dynamic and static calls executed at runtime, producing larger code regions for optimization.

Specific contributions of this paper include:

- A cloning algorithm for object-oriented languages which removes dynamic dispatches resulting from parametric polymorphism while minimizing the number of clones.
- An empirical evaluation of the efficiency and effectiveness of the cloning techniques on a suite of program.

The remainder of the paper is organized as follows. In Section 2 we describe the difficulties of optimizing object-oriented programs and introduce our compilation framework. Section 3 describes how global information is enhanced through cloning and how the number of clones is minimized. In Section 4 we report the results of our application of these techniques. Related work is discussed in Section 5 and we summarize in Section 6.

2 Background

In this section we examine characteristics of object-oriented programs which affect their efficiency. Then we briefly discuss the flow analysis techniques from which the cloning algorithm proceeds.

2.1 Efficiency of OOP and COOP Languages

Object-oriented programming provides tools for data abstraction and type-dependent dispatch, supporting both increased program modularity and code reuse. It supports polymorphism, late binding (dynamic dispatch or virtual functions calls), and inheritance, enabling programmers to organize their programs hierarchically as special cases based on general solutions, and to hide the details of operations. Likewise, concurrent object-oriented programming enables programmers to abstract and encapsulate consistency mechanisms, parallelization and

data layout decisions. This, in turn, makes the programs easier to understand and modify. Object-oriented programs differ greatly in structure from procedural code [5], and there is every indication that these differences increase as programmers develop an "object-oriented" programming style.

Though they have desirable software engineering advantages, OOP and COOP typically have an adverse impact on performance. Due to high levels of hardware parallelism (deep pipelines and multiple issue), modern processors are heavily dependent on instruction scheduling and register allocation to achieve good performance. However, these optimizations require unbroken sequences of instructions, or, at the very least, good control flow information. Since object-oriented programs tend to have very small functions, inlining is required to enable these optimizations. Unfortunately, dynamic dispatch confounds control flow, seriously complicating or preventing inlining. Likewise, parallelization, data layout, blocking and other high level transformation rely on interprocedural control flow information [19].

To inline functions a compiler for object-oriented languages must know the exact type of an object (as opposed to the declared type of which it may be a subclass). This *concrete type information* is precisely the detail the programmer wishes to hide, via encapsulation and code reuse. Concrete types can be used to generate efficient code sequences which manipulate the representations of data types. For example, a sort algorithm is described in terms of comparing and moving elements. However, comparing and moving numbers as opposed to character strings, is very different and subject to different optimizations. In the absence of concrete type information, an implementation must use run time checks or dynamic dispatches, which can lead to poor performance.

2.2 Polymorphism

Polymorphism refers to the ability of a function or variable to operate on or contain objects of different types. We are concerned with two types of polymorphism: *parametric* and *true*. Parametric polymorphism occurs when a function invocation or instance of a class can be parameterized by types it uses, much like templates in C++. One popular use of parametric polymorphism is for "container" classes for sets, lists, hash tables etc. True polymorphism occurs when a specific function invocation or object contains a single reference which might be of more than one type. A typical use of such polymorphism is in a simulator, where the configuration of simulated elements is data dependent and cannot be determined at compile time. Our algorithm eliminates the parametric polymorphism exposed by the analysis which, as we will see in Section 4, is a major cause of dynamic dispatch. It is important to note that true polymorphism often cannot be eliminated since it represents a choice point in the program which would require a `case` or `switch` statement in a procedural language.

2.3 Global Program Analysis

In many object-oriented programs, the information necessary for optimization is still present in the program structure, but it is divided across module boundaries or even compilation units. Global program analysis can be an efficient and effective way of recovering information such as global control flow, global data flow, and concrete type information. In the sorting example, the type of data being

sorted may not be specified at the definition of the sort, but is determined at call site of a sort operation. Global analysis recovers this concrete type information, linking the caller and callee and breaking through abstraction boundaries to enable optimization.

Recently, global program analysis frameworks have been developed for object-oriented languages which can efficiently derive global control flow and concrete type information [27, 26, 1, 28]. These algorithms simultaneously infer the interwoven global control and data flow of object-oriented programs. They do so by a combination of flow analysis and abstract interpretation and by modeling the different environment in which a function is invoked by a set of "contours" [31]. Typical analyses create for each function a number of contours polynomial in the size of the program. Moreover, these contours often do not represent useful optimization opportunities.

2.4 Implementation Context and Applications

This cloning algorithm has been implemented in the Illinois Concert system which includes a complete development environment for irregular parallel applications. The Concert system supports a concurrent object-oriented programming model and includes a globally optimizing compiler, efficient runtime, symbolic debugger, and an emulator for program development. This system compiles ICC++ [18], a parallel C++ dialect, and Concurrent Aggregates [11, 10] for execution on the Cray T3D [15] and Thinking Machines CM-5 [32] as well as uniprocessor workstations.

Cloning is used in our system to enable unboxing and register allocation of integer and floating point numbers, unboxing of integer and floating point arrays, and inlining and static binding of functions, enabling us to obtain sequential efficiency comparable to C [30]. On parallel machines, the more precise control flow information has enabled us to specialize the calling conventions in our hybrid execution model [29]. We are also in the process of using it to create and optimize call graph subtrees based the location of data in parallel machines.

3 The Cloning Algorithm

The idea of cloning is to create specialized versions of data structures and methods (which we call concrete types and clones respectively) for the different ways in which they are used by the programmer. These versions are then shared across the program by ensuring that the appropriate one is called for each use. The cloning algorithm starts with the results of global analysis. First, we describe the pertinent information provided by this analysis. Next, we present a modified dynamic dispatch mechanism for finding the appropriate clones. Then, we show how to select clones to maximize optimization opportunities and ensure that the resulting call graph is realizable via the dispatch mechanism. Once the clones have been selected they are created by constructing new concrete types, duplicating methods and rebuilding the call graph including the dispatch tables.

3.1 Contours and Clones

Flow analysis of object-oriented programs produces information about data flow values for methods based on the contours (calling environments) in which they

are invoked and for instance variables based on the statement and contour at which they were created [26, 28]. We will call the contours for methods *method contours* and the statement and method contour at which objects with distinguished instance variables are created *class contours*.[2] Since these contours were created by the analysis to distinguish potentially different uses of methods or classes they roughly correspond to potential clones and concrete types. However, the analysis may distinguish method contours by any aspect of the calling environment including the contours from which they were invoked [26], the types of all the arguments [1] as well as other criteria [25]. Thus, a call graph on the contours cannot, in general, be realized by the standard dispatch mechanism.

3.2 Modified Dynamic Dispatch Mechanism

Cloning modifies the call graph by replicating subgraphs the methods of which are then called by only a subset of the previous callers. If a call site is statically bound (resolves to a single target method) it can be connected directly to the appropriate clone. However, if the call site requires a dynamic dispatch, the standard dispatch mechanism used by C++ or Smalltalk is, in general, insufficient to distinguish the correct callee clone. The problem is that this dispatch mechanism determines the method to be executed based on the selector (virtual function name) and runtime class of the target object $< selector, class >$, and these are identical for all clones of a given method. The example in Figure 1 illustrates this limitation.

```
class Stream;
class StringStream : Stream;
class Shape;
class Square : Shape;
class Circle : Shape;

Stream::print(Shape * o) { ... }

CLONE Stream:print(Square * o) { ... }
CLONE Stream:print(Circle * o) { ... }
CLONE StringStream:print(Square * o) { ... }
CLONE StringStream:print(Circle * o) { ... }
```

```
main() {
  Object * o = new Circle;
  Stream * s;

  if (...) s = new StringStream;
    else s = new Stream;
  s->print(o);
  o = new Square;
  s->print(o);
  ...
}
```

Fig. 1. Limitation of Standard Dispatch Mechanism

In Figure 1 the `print()` method in the `Stream` class takes a single argument `o` which is either a `Circle` or a `Square`. Since the variable `s` can be either a `Stream` or a `StringStream`, the invocation requires dynamic dispatch. However, the the standard dispatch mechanism only dispatches on the selector and the class of the target, and hence cannot select between the versions of `Stream::print()` cloned based on the type of parameter `o` (one for `Square` and one for `Circle`). Thus, a more powerful dispatch mechanism is required.

To address this problem we propose a call site specific dispatch mechanism. Each call site is given an identifier which is used during dynamic dispatch to distinguish the appropriate callee clone for each selector and target object type pair. In our example, the call site information would allow us to select the version of `print` for `Circle` at the first call site and that for `Square` at the second.

[2] Method contours and class contours correspond to entry sets and creation sets in [28].

Since only a single dimension is added, this mechanism is the smallest extension sufficient to select the correct clone, and, unlike multiple-dispatch, is independent of the number of arguments.

Cloning partitions the objects in user defined classes into concrete types which have more precise type signatures. From the point of view of the dispatch mechanism these are identical to user defined classes. Thus the modified mechanism uses the *concrete type* of the target object instead of the *class* during dispatch. Since the concrete type must be available at run time, objects are tagged when they are created with their concrete type (just as they would have been tagged with their class). Thus, the final modified dispatch mechanism uses $< call\ site, selector, concrete\ type >$ to select the method to be executed. Even if using this mechanism incurs additional overhead[3], the number of dynamic dispatches is greatly reduced, more than compensating for a slightly higher resolution cost.

3.3 Selecting Clones

Clones are selected by partitioning method and concrete types by partitioning class contours.[4] The initial set of partitions is determined by optimization criteria such as minimization of dynamic dispatch or unboxing. These partitions represent concrete types and versions of methods (clones) amenable to special optimization. Then, we iteratively refine the partitions until the cloned call graph is realizable by the dispatch mechanism.

```
clone_selection() {
  initial_method_contour_partition =  new Partition;
  initial_class_contour_partition = new Partition;
  forall m in method_contours do
    m.partition = initial_method_contour_partition;
  forall c in class_contours do
    c.partition = initial_class_contour_partition;
  while (!fixed_point) {
    repartition(method_contours,
                method_contours_equivalent);
    check_class_contours_required_for_dispatch();
    repartition(class_contours,
                class_contours_equivalent);
  }
}
```

```
repartition(set,equivalent){
  result = new Set;
  result.add( new Set(set.first()));
  forall e in set.rest() do
    forall s in result do
      if (forall r in s do
          equivalent(e,r))
        s.add(e);
      else result.add( new Set(e));
}
```

Fig. 2. Cloning Selection Drivers (pseudocode)

The overall algorithm is presented in Figure 2. It is based on two functions, one which determines if two method contours can share a clone (are *equivalent*) and another for class contours. Using these functions (shown in Figure 3) we first compute a partition of method contours then compute a partition of class contours. The `repartition` function for partitions by grouping the contours such that all the contours in a partition are equivalent. Since a finer partition of class contours can induce a finer partition of method contours (to ensure realizability)

[3] The modified dispatch mechanism is amenable to optimizations such as folding the call site id into the selector to form a single index into the virtual function table, or the use of multi-dimensional dispatch tables.

[4] Some analyses use contours which cannot be differentiated by our modified dispatch mechanism. For such analysis a set of minimum partitions is precomputed.

and vice versa we repeat the process until a fixed point is reached. Since the number of contours is finite and the partitioning proceeds monotonically (see Figure 3 under the comment `monotonicity`) termination is ensured.

```
boolean method_contours_equivalent(a,b) {
 return
     &&((a.partition == b.partition)                    /* monotonicity */
     && (foreach s in callsites(method(a)) do           /* optimization criteria */
          binding(s,a)==binding(s,b))
     && (foreach v in variables(method(b)) do
          boxing(v,a)==boxing(v,b))
     && (foreach c in creation_points(method(a)) do     /* realizability */
          class_contour(c,a)==class_contour(c,b));
}

boolean class_contours_equivalent(a,b) {
 return
       ((a.partition == b.partition)                   /* monotonicity */
     && (foreach v in instance_variables(class(b)) do  /* optimization criteria */
          boxing(v,a)==boxing(v,b))
     && (! b in a.not_equivalent);                     /* realizability */
}

check_class_contours_required_for_dispatch() {
  foreach s in callsites do
    foreach e1,e2 in call_graph_edges(s) do
      if  ((method_partition(e1.callee) != (method_partition(e2.callee)))
        && (e1.selector == e2.selector)
        && (class_partition(e1.target) == (class_partition(e2.target))))
        make_not_equivalent(class_contour(e1.target),
                            class_contour(e2.target));
}

make_not_equivalent(a,b) {
  a.not_equivalent.add(b);
  b.not_equivalent.add(a);
}
```

Fig. 3. Contour Equivalence Functions (pseudocode)

The initial partitions are built based on optimization criteria used by the contour equivalence functions. For example, to maximize static binding we examine each call site in the method for the two contours, and if they would bind to different clones (method contour partitions) or different sets of clones we declare the two contours not equivalent. Similarly for representation optimizations, if a variable within two method contours or an instance variable within two class contours has different efficient representations (unboxed or inlined objects) grouping the contours would prevent optimizations, so we declare them not equivalent. The code to check these optimization criteria appears in Figure 3 under the comments: `optimization criteria`. Standard techniques for profiling or frequency estimation [34] can be used to maximize the benefits of optimization while limiting code expansion.

To ensure that the call graph is realizable by the modified dispatch mechanism, further refinement of the partitions may be required. This affects both method and class partitions. The dispatch mechanism uses concrete type (class contour partition) to select the target method, so call sites can require two class contours to be in different partitions in order be able to resolve the appropriate method. This occurs when the $< call\ site, selector >$ pair does not resolve to a

unique target clone (method contour partition). For example, in Figure 4 we have decided to optimize the binding of `print()` in the method `print_contents()` to `Circle::print()` for circle containers and `Square::print()` for square containers. Now, at `site 3` the dispatch mechanism would like to select the appropriate specialized versions. Since the call site and selector are identical, it must use the concrete type of `c` to distinguish the correct version. Thus, the method contour partition of `print_contents()` has induced a class contour partition of `Container` to distinguish those instances for which `o` is a `Circle` from those for which `o` is a `Square`. The function which checks this condition and ensures that two class contours will be non-equivalent is `check_class_contours_required_for_dispatch` in Figure 3.

```
class Container { Object * o; ... };
void Container::print_contents(){ this->o->print(); }
Container * create() { return new Container; }

main() {
  Container *a = create(); /* site 1 */
  Container *b = create(); /* site 2 */
  a->o = new Circle;
  b->o = new Square;
  Container *c = a;
  if (...) c = b;
  c->print_contents();     /* site 3 */
}
```

Fig. 4. Example Requiring Repartitioning of Contours

Similarly, class contour partitions can induce method contour partitions. Class contours are defined by their creation point (creating statement and surrounding method contour). Since the partitions of class contours will be the concrete types which are used by the dispatch mechanism, objects must be tagged at their creation points with their concrete type. This means that two method contours cannot be in the same partition if they define different class contour partitions. For example, in Figure 4, we have partitioned the class contour for `Container` based on the type of `o` (`Circle` or `Square`). In order to tag circle containers and square containers as different concrete types, enabling the dispatch mechanism to select between them, we must repartitioning the method contours for `create()`, separating those called from site **2** from those called from site **3**. Thus, the class contour partition of `Container` has induced a method contour partition of `create()`. This is checked by the function `method_contours_equivalent` under the comment `realizability` in Figure 3.

3.4 Making Clones

When the fixed point is reached, we create method clones for the method contour partitions and concrete types for the class contour partitions. For each method clone, we duplicate the code and update the data flow information so that it reflects only the information stored in the contours for its partition. The call sites and variables will now have the more precise information dictated by the optimization criteria. Statically bound call sites are connected to the appropriate clone and are now amenable to inlining. Methods which contain creation points are

modified so that the created objects are tagged with the appropriate concrete type instead of the original class. Finally, the modified dispatch tables are constructed. Call sites which require dynamic dispatch are assigned identifiers. For each edge in the interprocedural call graph from these sites, an entry is made into the dispatch table mapping the $< call\ site, selector, concrete\ type >$ to the appropriate clone.

4 Experimental Results

We have implemented these cloning techniques in the Illinois Concert compiler and tested them on tens of thousands of lines of Concurrent Aggregates programs [10]. In this section we present results from a representative sample of those programs. These test programs are concurrent object-oriented codes written by a variety of authors of differing levels of experience with object-oriented programming. They range in size from kernels to small applications. They all make use of code sharing through polymorphism, and several also contain true polymorphism, for example using dynamic dispatch (instead of conditional tests) to differentiate data dependent situations.

Program	ion	network	circuit	pic	mandel	tsp	richards	mmult	poly	test
User Lines	1934	1799	1247	759	642	500	378	139	49	39
Total Lines	2384	2249	1697	1209	1092	950	828	589	499	489

The first three programs simulate the flow of ions across a biological membrane (**ion**), a queueing network **network** and an analog circuit (**circuit**). **pic** performs a particle-in-cell calculation, and **man** computes the Mandelbrot set using a dynamic algorithm. The **tsp** program solves the traveling salesman problem. **richards** is an operating system simulator used to benchmark the SELF system [8, 24]. The last three programs are kernels representing uses of polymorphic libraries. **mmult** multiplies integer and floating point matrices, **poly** evaluates integer and floating point polynomials and **test** is a synthetic code which uses multi-level polymorphic data structure. All the programs were compiled with the standard CA prologue of 450 lines of code.

4.1 Clone Selection

To evaluate, the clone selection algorithm we generated initial contour partitions using optimization criteria for removing all dynamic dispatches resulting from parametric polymorphism regardless of the number of invocations. In addition, we optimized the representation of all arrays and local integer and floating point variables by unboxing. We applied these criteria to cloning of our test suite and evaluated the number of concrete types and method clones produced.

In order to demonstrate that clone selection was able to combine contours not required for optimization we report the number of contours produced by our analysis. However, it should be noted that the number of contours produced by an analysis is only superficially related to the quality of information it produces and the difficulty of selecting clones based on that information. In theory, flow analyses produce $O(N)$, $O(N^2)$, $O(N^6)$ or more contours for a program of size N [27, 26, 1, 25]. The number of contours seen in practice can require large amounts of space [2]. The particular analysis we use is an iterative algorithm which creates

contours in response to imprecisions discovered in previous iterations [28]. As a result, it is much more conservative in the number of contours it creates than other analyses.

Selection of Concrete Types The number of user classes, analyzed class contours, and the number of concrete types produced by the selection algorithm are reported below:

Program	ion	network	circuit	pic	mandel	tsp	richards	mmult	poly	test
Program Classes	11	30	15	11	11	12	12	7	6	10
Class Contours	64	43	30	27	26	17	27	13	17	18
Concrete Types	11	32	15	11	11	12	13	7	6	10

The data shows that the number of class contours is much greater than the number of user-defined classes. However, the number of concrete types finally selected is closer the number of user classes. This is because not all those distinguished by the analysis are required for optimization. In particular, when all invocations on objects corresponding to some class contour are statically bound, the dispatch mechanism does not need a concrete type for dispatch and no distinct concrete type is created. Methods for such objects are simply specialized for the class contour and statically bound.

Selection of Method Clones The number of user defined methods actually used in the program (as determined by conservative global flow analysis), analyzed method contours, clones selected by our algorithm, and the final number of methods after inlining appear below. The inlining criteria is based on the size of the source and target methods as well as simple static estimation of the call frequency. When all calls to a method are inlined, that method is eliminated from the program.

Program	ion	network	circuit	pic	mandel	tsp	richards	mmult	poly	test
User Methods	348	330	143	157	108	103	129	48	42	40
Method Contours	720	555	511	271	168	153	280	139	189	87
Clones Selected	445	342	173	195	115	108	138	64	54	40
Clones After Inlining	347	181	101	148	63	71	65	42	26	22

Again, the analysis creates many more method contours than user defined methods. However, the selection algorithm chooses only those required for optimization; in most cases ending with only somewhat more than the number of user defined methods. Moreover, since many call sites can be statically bound after cloning, many of the smaller methods can be inlined at all their callers. Thus, the number of clones which remain after inlining is actually smaller than the number of methods in the original programs.

Code Size One important measure of the effectiveness of clone selection is the final code size. Figure 5 compares the resulting code size before and after cloning.

The cloned programs usually increase in size by some modest amount, and always by less than 70%. The relatively large increase in **ion** is the result of extensive use of first class selectors (virtual function pointers in C++) during the output phase of the program. Code size expansion can be reduced by using profiling or frequency estimation to restrict cloning to the parts of the program which execute the most. Since the output phase is only executed once, such restrictions would have helped for **ion**.

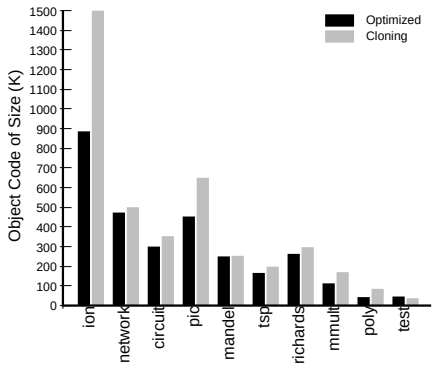

Fig. 5. Effect of Cloning on Code Size

4.2 Effect on Optimization

We evaluated the impact of cloning on optimization through its effect on the static and dynamic counts of dynamic dispatch as well as the total number of calls. We three different runs of our compiler. The base case *baseline* copies out inheritance (customization [6]) but does no cloning and inlines only accessors and operations on primitive types (like `integer`s and `float`s). This corresponds roughly to the optimization level for a hybrid language like C++. The *optimized* version includes global flow analysis and inlining and the *cloning* version includes the analysis, cloning and inlining.

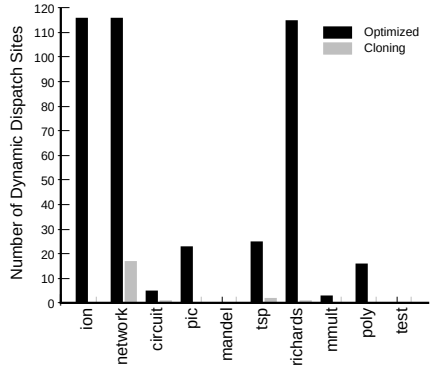

Fig. 6. Dynamic Dispatch Sites In Code

Dynamic Dispatch Sites Static binding is the process of transforming dynamic dispatches (virtual function calls) into regular function calls. Cloning enables static binding by creating versions of code specialized to the types they operate on. In Figure 6 we report the number of dynamic dispatch sites in the final

code. Without cloning all the programs but two (**mandel** and **test**) contain a number of dynamic dispatch sites. **mandel** is primarily numerical with only token polymorphism and in **test** the selectors (virtual functions) have unique names, enabling them to be statically bound even without analysis. With cloning, only one program has more than two dynamic dispatch sites. Those dispatches which remain correspond to the true polymorphism in the programs, and cannot be statically bound to single methods. For instance, in **richards** (the OS simulator) the single remaining dispatch is in the task dispatcher, where the simulated task is executed. Since the set of tasks is data dependent, this dynamic dispatch cannot be eliminated.

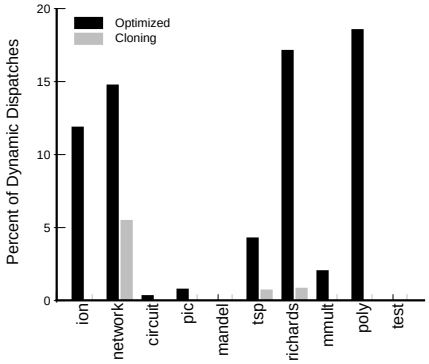

Fig. 7. Percent of Total Dynamic

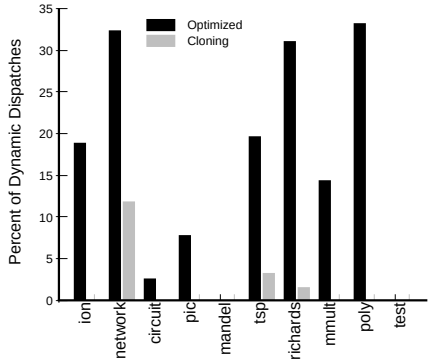

Fig. 8. Percent of Remaining Dynamic

Dynamic Dispatch Counts The runtime counts in Figure 7 demonstrate the effectiveness of cloning for elimination of dynamic dispatch during program execution. We ran our test suite using sample input and collected the number of calls executed, both static and dynamic. We report the number of dynamic dispatches as a percentage of those occurring in the *baseline* code. While global analysis and optimization alone is able to statically bind many calls, cloning is able to statically bind many more. Moreover, once the number of calls is reduced by inlining, those remaining in the *optimized* case are frequently dynamic dispatches. Figure 8 isolates the number of dynamic dispatches as a percentage of the remaining invocations. This shows that optimization of the *optimized* code is largely limited by dynamic dispatches which inhibit inlining. In contrast, cloning keeps that number to a tiny fraction of the total calls. Note that this graph should not be used to compare the absolute number of dynamic dispatches since the total number of calls in the cloned version is less than that in the optimized version.

Number of Calls In Figure 9 we report the total number of calls (static and dynamic) after optimization. For the baseline (100%) we use the number of calls in the `baseline` version. Global analysis and inlining eliminate between 35% and 99% of the calls, and in some cases cloning eliminates 20% more. We expect that better use of frequency information (which in our current compiler is limited),

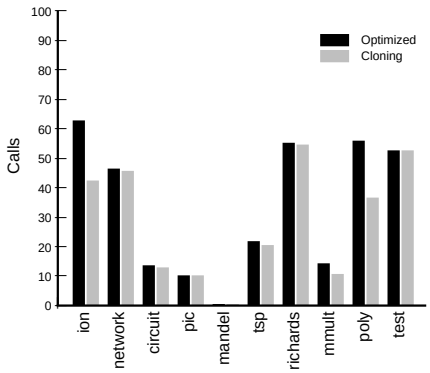

Fig. 9. Total Number of Calls

combined with the greater number of statically bound methods in the *cloning* version will enable us to reduce the number of calls even further.

5 Related Work and Discussion

Cooper [12] presents general interprocedural analysis and optimization techniques. Whole program (global) analysis is used to construct the call graph and solve a number of data flow problems. Transformation techniques are described to increase the availability of this information through linkage optimization including cloning. However, this work does not address clone minimization. Cooper and Hall [19, 21, 13, 14, 20, 22] present comprehensive interprocedural compilation techniques and cloning for FORTRAN. This work is general over forward data flow problems, and presents mechanisms for preserving information across clones and minimizing their number. However, concrete types are not a forward data flow problem. Hall determines initial clones by propagation of *clone vectors* containing potentially interesting information which are merged using *state vectors* of important information into the final clones. We handle forward flow problems in a similar manner, but rely on global propagation to determine the final clones for recursive functions.

Several different approaches have been used to reduce the overhead of object-orientation. *Customization* [6] is a simple form of cloning whereby a method is cloned for each subclass which inherits it. This enables invocations on `self` (or `this` in C++ terminology) to be statically bound. Another simple approach is to statically bind calls when there is only one possible method [3]. This idea was extended by Calder and Grunwald [4] through 'if conversion', essentially a static version of polymorphic inline caches [23]. Our work also shares some similarities with that done for the SELF [33] and Cecil [9] languages. Chambers and Ungar [7], used *splitting*, essentially an intraprocedural cloning of basic blocks, to preserve type information within a function. Early work on Smalltalk used inline caches [17] to exploit type locality. Hölzle and Ungar [24] have shown the information obtained by polymorphic inline caches can be used to speculatively inline methods. While run time tests are still required, various techniques are presented to preserve the resulting type information. None of these approaches uses globally

analyzes and transformation to eliminate the run time checks nor to preserve general global data flow information. More recently, Dean, Chambers, and Grove [16] have used information collected at run time to specialize methods with respect to argument types. While this can remove dynamic dispatches across method invocations, it does not handle polymorphic instance variables. Finally, Agesen and Hölzle have recently used the results of global analysis in the SELF compiler [2]. However, the information for all the contours for each customized method is combined before being used by the optimizer.

The cloning algorithm we have presented is general enough to enable optimization based on any data flow information provided by global flow analysis. All that is required is that the contour equivalence functions be modified to reflect the new optimization criteria. We have used optimization criteria for increasing the availability of interprocedural constants successfully with our cloning algorithm. However, efficient cloning for such information requires estimating its potential use for optimization which we have not yet implemented. Interested readers are referred to [19] for a discussion of the issues.

6 Summary and Future Work

Object-oriented programming is rapidly becoming a standard in program development. Traditional optimization techniques are severely hampered by the small methods and data dependent control flow of object-oriented programs. Cloning techniques can help resolve these problems, enabling object-oriented programs to achieving good performance on modern processors. We have shown that cloning can be used to eliminate dynamic dispatch and reduce the number of function calls. In effect, this removes the overhead of object-orientation, by enabling the compiler to undo the effects of information hiding and code sharing. We have demonstrated the effectiveness of cloning for optimization on a collection of object-oriented programs. We have also shown that the benefits can be achieved at modest cost; the code size growth required to accrue full optimization potential is relatively small.

To continue this work, we are examining alternatives for extending the idea of equivalence of portions of storage maps of concrete types across classes. This will allow further clone elimination, removing additional redundancies in the final code. We are also examining optimization opportunity estimation metrics for cloning with respect to other types of data flow information.

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 and Urs Hölzle. Type feedback vs. concrete type analysis: A comparison of optimization techniques for object-oriented languages. Technical Report TRCS 95-04, Computer Science Department, University of California, Santa Barbara, 1995.
3. Apple Computer, Inc., Cupertino, California. *Object Pascal User's Manual*, 1988.
4. 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.
5. Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying differences between C and C++ programs. Technical Report CU-CS-698-94, University of Colorado, Boulder, January 1994.
6. C. Chambers and D. Ungar. 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.
7. 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.

8. 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.
9. Craig Chambers. The Cecil language: Specification and rationale. Technical Report TR 93-03-05, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, March 1993.
10. Andrew A. Chien. *Concurrent Aggregates: Supporting Modularity in Massively-Parallel Programs*. MIT Press, Cambridge, MA, 1993.
11. 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.
12. 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.
13. K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the IEEE Computer Society 1992 International Conference on Computer Languages*, pages 96–105, April 1992.
14. K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–118, April 1993.
15. Cray Research, Inc., Eagan, Minnesota 55121. *CRAY T3D Software Overview Technical Note*, 1992.
16. Jeffrey Dean, Craig Chambers, and David Grove. Identifying profitable specialization in object-oriented languages. Technical Report TR 94-02-05, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, February 1994.
17. L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Eleventh Symposium on Principles of Programming Languages*, pages 297–302. ACM, 1984.
18. The Concurrent Systems Architecture Group. The ICC++ reference manual, version 1.0. Technical report, University of Illinois, Department of Computer Science, 1304 W. Springfield Avenue, Urbana, Illinois, 1995. Also available from http://www-csag.cs.uiuc.edu/.
19. M. W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, 1991.
20. M. W. Hall, S. Hiranandani, and K. Kennedy. Interprocedural compilation of Fortran D for MIMD distributed memory machines. In *Supercomputing '92*, pages 522–535, 1992.
21. Mary W. Hall, Ken Kennedy, and Kathryn S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of the 4^{th} Annual Conference on High-Performance Computing (Supercomputing '91)*, pages 424–434, November 1991.
22. 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.
23. Urs Hölzle, Craig Charmbers, 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.
24. 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.
25. 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.
26. N. Oxhøj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *Proceedings of OOPSLA '92*, 1992.
27. J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA '91*, pages 146–61, 1991.
28. John Plevyak and Andrew A. Chien. Precise concrete type inference of object-oriented programs. In *Proceedings of OOPSLA*, 1994.
29. 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.
30. 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.
31. Olin Shivers. *Topics in Advanced Language Implementation*, chapter Data-Flow Analysis and Type Recovery in Scheme, pages 47–88. MIT Press, Cambridge, MA, 1991.
32. Thinking Machines Corporation, 245 First Street, Cambridge, MA 02154-1264. *The Connection Machine CM-5 Technical Summary*, October 1991.
33. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA '87*, pages 227–41. ACM SIGPLAN, ACM Press, 1987.
34. 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.

This article was processed using the LaTeX macro package with LLNCS style