Field Analysis: Getting Useful and Low-cost Interprocedural Information

Sanjay Ghemawat [*]
Systems Research Center
Compaq Computer Corp.
sanjay@google.com

Keith H. Randall
Systems Research Center
Compaq Computer Corp.
randall@pa.dec.com

Daniel J. Scales
Western Research Laboratory
Compaq Computer Corp.
scales@pa.dec.com

Abstract

We present a new limited form of interprocedural analysis called *field analysis* that can be used by a compiler to reduce the costs of modern language features such as object-oriented programming, automatic memory management, and run-time checks required for type safety. Unlike many previous interprocedural analyses, our analysis is cheap, and does not require access to the entire program. Field analysis exploits the declared access restrictions placed on fields in a modular language (*e.g.* field access modifiers in Java) in order to determine useful properties of fields of an object.

We describe our implementation of field analysis in the Swift optimizing compiler for Java, as well a set of optimizations that exploit the results of field analysis. These optimizations include removal of run-time tests, compile-time resolution of method calls, object inlining, removal of unnecessary synchronization, and stack allocation. Our results demonstrate that field analysis is efficient and effective. Speedups average 7% on a wide range of applications, with some times reduced by up to 27%. Compile time overhead of field analysis is about 10%.

1 Introduction

Modern languages such as Java and Modula-3 provide features such as type safety, object-oriented method dispatch, and automatic memory management that improve programmer productivity, reduce bugs, and improve security. However, because of the run-time cost of these features, applications written in these languages and compiled with standard optimizations are often slower than similar applications in languages such as C and Fortran. For example, type safety requires that all array references include bounds checks. Virtual method calls are more expensive than direct calls to procedures. Allocating an object on the heap typically introduces more overhead than stack allocation of the object.

These overheads can often be eliminated by compiler optimizations. For example, a bounds check can be eliminated if the compiler can prove that the index of the array reference is non-negative and less than the length of the array. A virtual method call can be converted to a direct call if the compiler can prove that there is only one possible target method. Similarly, a new object can be allocated in a stack frame if, among other things, the compiler can show that a reference to the object never escapes to another thread or to a method higher in the stack.

Compiler analyses for these optimizations traditionally have been whole-program analyses such as *class hierarchy analysis* (CHA), where the entire set of classes is examined to determine the exact class hierarchy [7], or some type of interprocedural dataflow analysis. Both of these methods can be quite expensive in terms of compile time because they must examine many classes simultaneously.

We have been investigating *field analysis*, a cheaper form of interprocedural analysis that is applicable to modular, object-oriented languages. In particular, the scope of analysis is a single class or a limited set of classes, and no form of interprocedural dataflow analysis is required. Field analysis is therefore less dependent on an ability to determine the exact call graph, which is often difficult in object-oriented applications. Because of these properties, field analysis is cheap enough to be used in a compiler which is invoked dynamically at run-time on frequently executed methods.

Field analysis determines useful properties of a field of an object by scanning the code that could possibly access that field, as dictated by language access rules. For example, a `private` field in Java can only be accessed by methods in the local class, while a `package` field[1] can only be accessed

[*] Author's current affiliation is Google, Inc., 2400 Bayshore Parkway, Mountain View CA 94043

[1] A Java field without any access modifiers is visible to its entire package and therefore we call it a `package` field.

by methods in classes that are in the same package as the containing class. An example of a field property is more exact information on the type of a field. Field analysis can often show that a field, if non-null, only references objects of class C, and never objects of any of C's subclasses. Similarly, field analysis can often prove that a field, if non-null, always contains an array of a particular constant size.

These simple field properties can be used to attack a variety of overheads. Exact type information can help resolve virtual method calls, and other properties can be used to prove that null checks and bounds checks are unnecessary. Field analysis can be used in addition to or as an alternative to optimizations like CHA that assume that all classes (i.e. all application code) are known at compile time. Performing CHA is not always desirable, because if it is used in cases when additional classes may be loaded later, then it may be necessary to invalidate code that has used the CHA assumption. Field analysis is therefore useful when loading the entire class hierarchy is too expensive or it is undesirable to invalidate code later. Even if CHA with code invalidation is used, it is beneficial to apply field analysis first, since a method call resolved using field analysis will not have to be invalidated later.

In this paper, we describe our implementation of field analysis in the Swift compiler. Swift [18] is a complete optimizing Java compiler for the Alpha architecture which uses static single-assignment form for its internal representation. The compiler is written in Java and it translates Java bytecodes to Alpha code. We describe the basic set of field properties that are checked and their use in a number of simple optimizations. We then describe several more complex field properties, and their use in performing inexpensive analyses for object inlining, stack allocation, and synchronization removal. We then present performance results and statistics for a variety of optimizations enabled by field analysis on a number of applications, including the SpecJVM98 applications.

2 Basic Field Analysis

In this section, we describe an implementation of field analysis and a basic set of useful field properties. Our approach to field analysis is to scan all the code that can access a field and then deduce properties of the field from the properties of all the accesses to the field. Field analysis is therefore applicable to any modular, object-based language. However, because the particular details and the useful properties depend on the language, we will describe our implementation for the Java language.

Most of the field properties that are currently checked are relevant only for reference fields (i.e. fields that contain references to objects or arrays, rather than *scalar types* such as integers). Therefore our current implementation ignores fields with scalar types.

Class	**Field**	Code to scan
`public`	`private`	containing class
`public`	`package`	containing package
`public`	`protected`	containing package and subclasses
non-public	`private`	containing class
non-public	non-private	containing package

Table 1: Field Access Regions

2.1 Finding All Accesses to a Field

We use Java field modifiers to determine the subset of the program that must be checked for accesses to a given field. This mechanism is summarized in Table 1. The first two columns contain the class and field modifier respectively and the third column describes the subset of the program that has to be scanned for accesses to a field with the specified modifiers. For a `final` field, the above rules are used for finding all the reads, but only the containing class has to be scanned for writes to the field.

Note that for a `protected` field in a public class `T`, we scan all subclasses of `T`. Dynamic loading could introduce new subclasses of `T`, and therefore our compiler analyzes such fields only if CHA is also being used (or if the class is declared `final`). `Public` fields in public classes could be handled by scanning the entire program, but Swift currently ignores them for efficiency considerations.

Dynamic loading could also potentially create problems in our handling of package-visible fields by introducing a new class into a package. However, the three most widely used class loaders, namely the system loader, the installed extensions loader, and the application loader, simply load from predetermined portions of the file system. Swift can therefore scan those portions of the file system to determine the base set of packages and can be sure that dynamic loading will not introduce any new classes into these packages. (We assume that the contents of these parts of the file system do not change between compilation time and run time.)

2.2 Computing Field Properties

Field analysis uses a table that contains information about the properties of the fields being analyzed. The Swift compiler does the analysis by converting each method being scanned from Java bytecode to its standard machine-independent intermediate representation (IR) based on static single-assignment (SSA) form. The SSA graph is essentially a factored use-def graph, and Swift also incrementally maintains def-use information for each node in the graph while building and modifying the graph. Once the SSA form is built, Swift iterates through all nodes in the graph which represent loads of fields or stores to fields and incrementally updates the information known about the accessed field. For each load of a field, it analyzes each use of the load. For each store of a field, it analyzes both the value stored into the

Property	Definition
exact_type(*field*)	the field is always assigned a value of the specified type
always_init(*field*)	the field is assigned exactly once in each constructor no constructor accesses the field before assigning it no constructor "leaks" this
only_init(*field*)	the field is not assigned anywhere outside a constructor
source_type(*field*)	an indication of what kind of values are assigned to the field: new, new/null, new/null/param, or other
uses_header(*field*)	the header of the object referenced by the field is possibly used
may_leak(*field*)	the object referenced by the field may be stored into the heap or returned by a method

nonnull(*field*)	exact_type(*field*).nonnull && always_init(*field*)
final(*field*)	always_init(*field*) && only_init(*field*)
inlineable_with_header(*field*)	final(*field*) && source_type(*field*) == new && exact_type(*field*) == static_type(*field*)
inlineable_without_header(*field*)	inlineable_with_header(*field*) && !uses_header(*field*) && !may_leak(*field*)
encapsulated(*field*)	!may_leak(*field*) && source_type(*field*) == new/null
never_leaks(*field*)	!may_leak(*field*) && source_type(*field*) == new/null/param

Table 2: Basic and Derived Properties of Fields

field and any other uses of that value. Because of the use of SSA form, the extracted properties are flow-sensitive. However, Swift only uses local information in proving properties and does not derive any context-sensitive properties. No optimizations are applied to the SSA form before it is analyzed.

Some of the field properties can be best expressed in terms of a generalized type for objects stored in a field. Swift has a simple but effective system of types which is used to summarize the possible results of nodes in an SSA graph or contents of fields of an object. The Swift type system includes the standard Java type system as a subset, but also allows specification of the following additional properties about a value with a particular Java type T:

- the value is known to be an object of exactly class T, not a subclass of T
- the value is an array with a particular constant size
- the value is known to be non-null

By incorporating these properties into the type system, we can describe important properties of any node in the SSA graph by its type. In addition, we can easily indicate properties for different levels of recursive types, such as arrays. One possible generalization of the type system is to allow union types. However, we have not found this extension to be very useful for the Java applications that we have examined.

While building the SSA graph, Swift automatically assigns the appropriate types to nodes representing method arguments, loads of fields of objects, and loads of global variables. In addition, it assigns non-null exact types to nodes representing newly allocated objects. It may also be able to give constant-size array types to nodes representing newly allocated arrays. Swift then does a standard type propagation to determine types for the remaining nodes in the graph. The type propagation provides extra information that is immediately usable for field analysis.

Building the SSA graph from bytecode involves building the CFG, calculating the dominator tree, determining phi-node placement [20], and doing abstract interpretation over the bytecodes to create and connect the SSA nodes. The dominator-finding algorithm [15] is super-linear in the worst case, but is in practice linear, and the remaining passes are linear. Hence, the analysis time is approximately linear in the amount of code to be analyzed, and requires storage only for the table of analyzed fields, plus the IR for a single method. For some properties (e.g. always_init, as described below), additional storage is required to save some information about methods, so that methods are not repeatedly analyzed.

2.3 Basic Properties

The basic properties of fields that we compute are shown in Table 2. The first property, exact_type, records the known information about the type of the values assigned to the field. Thus, exact_type can indicate that a field is always assigned values from a single class, values that are non-null, or values that are a constant-sized array. In our current implementation, we simply use the type of a node in the SSA graph to determine if it represents a value that is non-null, has an exact type, or has an exact array length. Hence, we will most often prove useful properties about the exact_type of a field if it is only assigned newly allocated objects. For efficiency, we do not currently try to prove that a value is non-null or an exact type in any other way. Note that exact_type describes the type of a field *after* its first assignment, so a non-null type does not necessarily imply that all accesses return non-null values.

The second property, always_init, is useful in proving that a field is always non-null. always_init indicates that the field has always been initialized (assigned) before it is accessed. To simplify analysis, we use somewhat conservative conditions. As shown, we require that the field is assigned exactly once in any normally-terminating call to a constructor for the containing class, and that no constructors access the field before it is assigned.[2] An operation will occur exactly once in a method call if the operation does not occur in a loop and it dominates the normal exit of the method. Additionally, we require that a reference to the new object (accessed via the this keyword) is not "leaked" in a constructor.

We consider this to leak if it is stored into the heap or passed to an unanalyzed method. If this is leaked before a field is initialized, then another thread (or even the current thread) may be able to use the reference to this to access the uninitialized state of the field. Because most constructors invoke superclass constructors, we recursively analyze all resolvable method calls with this as a receiver (which includes all superclass constructor calls, as well as others). However, it is not common in constructors to pass this as a non-receiver, so we do not recursively analyze methods calls which take this as a non-receiver argument. This restriction often limits the set of extra methods that need to be analyzed to those in the current class and its superclasses. Hence, this computation is very cheap, even though it may involve methods that are not in the local class or package.

The third property, only_init, indicates that a field is not assigned outside of a constructor. The remaining basic properties in Table 2 are used for our more advanced optimizations and will be described in later sections.

Table 2 also indicates some properties derived from these basic properties. For example, a field is always non-null when read if it is always_init *and* its exact_type property indicates that it is non-null. A field is final (unmodified after being initialized) if it is always_init and only_init.

As an example, consider the code in Figure 1. Because the field points is private, the compiler only needs to scan the instance methods in class Plane to determine its properties. It is easy to see that exact_type(points) must indicate a non-null array with base-type Point and a fixed size of 3. In addition, the basic properties always_init(points) and only_init(points) are both true, implying that the derived properties nonnull(points) and final(points) are both true as well.

[2] We also allow an assignment to a field to be implicitly performed by calling another constructor in the same class.

```
public class Plane {
  private Point[] points;

  public Plane() {
    points = new Point[3];
  }

  public void SetPoint(Point p, int i) {
    points[i] = p;
  }

  public Point GetPoint(int i) {
    return points[i];
  }
}
```

Figure 1: Example Class for Field Properties

2.4 Basic Optimizations

These field properties can be used in a number of simple optimizations. If the exact_type property of a field indicates that it contains objects only of a certain class, then a virtual method call whose receiver is the contents of the field can be resolved to a direct call. In the same way, exact_type information can be used to statically evaluate type-inclusion tests, such as instanceof or array store checks[3] in Java. If exact_type(*field*) indicates that field only contains arrays of a fixed size, then bounds checks and other expressions that use the length of the array can be simplified and often eliminated. If nonnull(*field*) is true, then any null check (required in Java for all non-static field and method accesses) on the contents of the field can be eliminated. While most Java systems can use page protection to implement null checks without any extra code, eliminating the null check is still useful because it gives the compiler more flexibility in doing code motion.

For example, in the SetPoint method in Figure 1, some possible optimizations include eliminating the null check on points, using the constant 3 in the bounds check computation, and eliminating the array store check.

Field properties can even make standard optimizations more effective. If final(*field*) is true, then *field* is known not to be modified in any method except the constructor for its containing class. A constructor can only be called immediately after allocating a new instance of an object, or by a constructor of a subclass. A method which is not a constructor cannot possibly call the constructor on a preexisting reference and so cannot possibly modify the final field of the reference. Therefore, if we have two loads of a field of the same object separated by method call which is not a constructor, we can apply common subexpression elimination to the two loads, even if we cannot analyze the method call.

[3] Storing to an array in Java requires a run-time check, called an *array store check* to ensure that the object being stored into the array is compatible with the base type of the array.

2.5 Limitations of Field Analysis

Our field analysis has a number of potential limitations. First, native methods cannot be analyzed by the compiler. The compiler therefore assumes that a native method reads and writes all fields that are accessible to a non-native method defined in the same class. (The compiler is augmented with built-in knowledge about the effects of some widely used native methods to make this analysis less conservative.) If any native method bypasses normal field protections, our field analysis will be incorrect and should be disabled on these fields.

The reflection facilities in Java can also be used to bypass field protections via the `setAccessible` method in the `java.lang.reflect` package. However, code must be explicitly granted permission to use `setAccessible` by the current security policy. Such bypassing of field protections is typically only used for system functions such as debugging or serialization of data for writing to disk. Again, our field analysis should be disabled on the fields in question. If the compiler is integrated with the run-time system, the implementation of `setAccessible` could potentially be modified to invalidate compiled code that has been optimized based on properties of the associated field.

Finally, the applicability of some optimizations may be affected by the Java memory model. In a multi-threaded program, a field may potentially be seen in an uninitialized state even if `always_init`(*field*) is true. Suppose one thread creates an instance of a class containing that field and stores a reference to the object in a global variable. On the local processor, the initializing write to the field precedes the write of the reference to the global variable. However, these writes are not required to appear in this order on a remote processor, unless there is proper synchronization on both processors.

Note that this problem can only occur on a multiprocessor with a weak memory consistency model for an object whose reference is stored in the heap without any synchronization. Under these conditions, it would be improper for safety reasons to remove a null check on the contents of the field for which `nonnull`(*field*) is true. However, it may be acceptable to give the compiler extra freedom in moving such null checks, since there is a data race between the initialization of and access to the object, so results are likely to be unpredictable anyway. Similarly, accesses to a field for which `final`(*field*) is true could yield an uninitialized value as well as the initialized value. The CSE optimization described above for final fields may be improper if the intervening method has synchronization, since the second load of the field might be required to see the initialized value of the field, according to the current Java memory model.

3 Object Inlining

We have also used the field analysis approach to determine fields that are candidates for *object inlining*. Object inlining [12] is a method of reducing the overhead of accessing objects by allocating the storage of an object within its containing object. If object B is inlined in object A, the cost for accessing object B is reduced by a pointer dereference. Also, garbage collection costs are reduced, since there is only a single allocation unit to be scanned, rather than two. Objects A and B are likely to be frequently accessed together, so cache locality may be improved by inlining if A and B are now stored in the same cache line. As an example of object inlining, Figure 2(a) shows the storage layout for a `Ray` object which references two `Point` objects. Figure 2(b) shows the storage layout when the two `Point` objects are inlined into the `Ray` object.

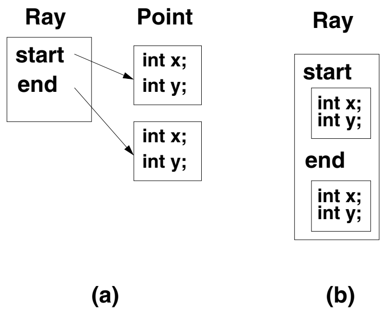

Figure 2: Example of Object Inlining

3.1 Analysis

Our analysis is unique in that it finds both objects that can be inlined without a header for the inlined object, and objects that can be inlined, but require an object header. The object header contains an indication of the type of the object and its method table, and is also typically used for synchronizing on the object. The header is therefore necessary if the program might execute certain operations, such as virtual method invocations, synchronization, and type-inclusion checks, on the object. In addition, if an object is inlined with a header, then a reference to the object can be allowed to escape into the heap. The garbage collector can determine via the header that a reference accesses an inlined object, and can therefore avoid collecting the containing object. We are not aware of any other implementations of object inlining that allow references to inlined objects to be stored in the heap. However, unlike some other systems, our approach is not context-sensitive, so we inline object B at field `f` of object A, only if it is provable that the contents of field `f` can be inlined for any instance of the class of object A.

By using the field analysis information described in Table 2, we can easily determine if the contents of a field can be inlined in the containing object. The first property that we use, `source_type`, indicates what is known about the source of the values stored into the field. It indicates that the field has only been assigned newly allocated objects (new), only new objects or null (new/null), only new objects, null, or method parameters (new/null/param), or none of these.

Initial code	Code after inlining field `f` in `y`'s class
`x = y.f;`	`x = y + offset(y, f);`
`x = new T;`	`x = y + offset(y, f);` if necessary, initialize header of `y.f`
`y.f = x;`	$< deleted >$

Table 3: Code Generation for Inlined Fields

(The new/null or new/null/param properties will be useful for escape analysis, as described in the next section.) In the case of object inlining, we are interested in fields which are only assigned newly allocated objects. The second property, `uses_header`, is used for determining if an inlineable object must contain the standard object header. The third property we use, `may_leak`, indicates that the contents of a field may "leak" out of the containing object by being stored into the heap or being returned by a method.[4]

Given these properties, we can specify when the contents of a particular field can be inlined, either without or with an object header, as shown in Table 2. If a field is to be inlined with a header, we require that the field be initialized with a single, final value which is a newly allocated object. For simplicity, we require that the class of the inlined object must be the same as the static type of the field. In this way, object layout is simpler for the underlying Java Virtual Machine (JVM), because inlined objects always have the same class as their static type. If a field is to be inlined without a header, we additionally require that the header of the object referenced by the field is never used and a reference to this object never leaks.[5]

Our analysis also allows us to inline a field that references an array. In that case, there is one extra condition (not mentioned in Table 2) that the allocated array must have a statically fixed constant size. Instances of the containing class (with the inlined array) then have a constant size and can be allocated using the normal object allocation mechanisms.

3.2 Implementation

Our analysis allows references to inlined objects to escape into the heap as long as the object includes a header. We have modified the underlying JVM to mark the headers of such inlined objects. The garbage collector can therefore determine when it encounters a reference to an inlined object. In such cases, the garbage collector scans from the start of the current virtual memory page to find the object that contains the referenced inlined object. It then preserves the storage for the entire enclosing object, not just the inlined part.

Code generation for inlined objects is fairly simple and the required transformations are listed in Table 3. A load of a reference to an inlined object is transformed into a simple address calculation. The additions in the transformed code can often be combined with other additions by standard compiler optimizations, so dereferencing an inlined object typically has zero cost. Code that initializes a field with an inlined object is transformed into code that initializes the header of the object, if necessary, and creates a reference to the inlined object. In our current implementation, the body of the inlined object does not have to be initialized separately from the enclosing object, because the `new` operation will return a zeroed block of memory. For an inlined multi-dimensional array, the inlined array may need to be initialized with references to the necessary subarrays. Any null checks on the contents of the inlined field are eliminated, since the field is guaranteed to be non-null at all accesses.

4 Escape Analysis

Escape analysis is used to determine if a reference to an object escapes a thread (i.e. can be accessed by another thread) or a particular method call (i.e. can still be accessed by the current thread after the call completes). Escape analysis is a necessary component for determining if an object can be allocated on the stack, rather than the heap. If a reference to an object does not escape a particular method call, then the object can be allocated on the stack frame of that call. Escape analysis can also be used for eliminating or reducing the cost of unnecessary synchronization. If a reference to an object does not escape a thread, then synchronization on the object is unnecessary.[6] In the discussion below, we will focus on determining if an object escapes a method call; proving that an object does not escape a thread involves only a little extra analysis related to the thread creation routines. For convenience, we will simply say that "an object escapes" when we mean that a reference to the object escapes a method call.

Escape analysis has typically involved either fairly simple analyses or highly complex and expensive computations. The simplest analysis [13] assumes that an object escapes if a reference to the object is ever stored into a global variable or a heap object (including arrays). For convenience, we will simply say in this case that the object is stored into the heap.[7] When a simple interprocedural analysis is used to determine the effects of method calls, this analysis is typically effective at finding many objects that don't escape. In contrast, other forms of escape analysis [5] do a full interprocedural and context-sensitive dataflow that builds up a "points-to graph"

[4] Our implementation of `may_leak` for object inlining is quite conservative but fast (it assumes that an object leaks if it is passed as a non-`this` argument to a method), whereas our escape analysis described below uses a more general implementation.

[5] This optimization could be generalized by allowing multiple assignments to the field, as long as all of the assigned values are new objects with the correct type, and the contents of the field are not involved in any pointer-equality comparisons.

[6] Under the current Java memory model, the synchronization cannot necessarily be removed, since the use of synchronization must cause all updates by the local processor to be committed and all updates by other processors to be seen. However, the Java memory model will likely be revised so that synchronization only has such effects for accesses that are connected by chains of synchronizations on the same objects [1].

[7] For the purposes of escape analysis, we also consider an object to be stored if it is thrown as an exception.

Property	Definition
`returns_unaliased`(*method*)	*method* returns a new, unaliased object or null
`may_return_param`(*method*, *i*)	input parameter *i* may be returned by *method*
`may_store_param`(*method*, *i*)	input parameter *i* may be stored into the heap or returned by *method*
`may_contain_param`(*method*, *i*)	the method may store a reference to input parameter *i* in a "never_leaks" field of the object referenced by param 0, but does not otherwise store parameter *i*
`may_store_param2`(*method*, *i*)	`may_store_param`(*method*, *i*) && !`may_contain_param`(*method*, *i*)

Table 4: Basic and Derived Properties of Methods

that summarizes the storage relationship of sets of objects. These kinds of analyses can prove that an object doesn't escape, even if a reference to the object is stored into the field of another object. However, these analyses can be very expensive in terms of computation time and memory, because of their manipulations of points-to graphs and their context-sensitive summaries of methods.

We have developed an extension of the simple analysis which uses some additional field properties to find more objects that don't escape. Because the field analysis is so cheap, our extension costs little more than the simple escape analysis, yet finds significantly more objects that don't escape. We first describe our implementation of the simple analysis, and then our extension using field properties.

4.1 Simple Analysis

Our simple escape analysis proceeds by finding a candidate value in a method M, and then proving that the value is never stored into the heap and is not returned by M. A candidate value is an object that is allocated directly in M, or a newly allocated object O returned by a method call in M such that O is known not to have been otherwise stored into the heap. To determine which methods return candidate values, the `returns_unaliased` property in Table 4 is calculated for each resolvable method call in M.

The compiler then proves that a candidate value does not escape by examining all uses of the value in M and checking that none return the value or store the value in the heap. When the candidate value is used in a method call, we use the second and third properties in Table 4 to determine if the value escapes. If the property `may_store_param` is true, the value is assumed to escape, and if the property `may_return_param` is true, the value is assumed to escape if the return value of the method call escapes (the truth of which is computed recursively).

The first three properties in Table 4 can be determined using a simple depth-first analysis of methods that makes the conservative assumption that an unanalyzable method call returns and stores all parameters and does not return an unaliased object. This assumption is also made for a call to a method that is currently being analyzed (i.e. for a recursive method call). Alternatively, a full interprocedural dataflow analysis that handles recursive method calls less conservatively could be used.

```
Pair p = new Pair();
Integer x = new Integer(5);
p.first = x;

class Pair {
  private Object first;
  private Object second;
}
```

Figure 3: An Encapsulated Field Example

```
Vector v = new Vector();
...
Enumeration e = v.elements();
while (e.hasMoreElements()) {
  Object o = e.nextElement();
  ...
}

class Vector {
  Enumeration elements() {
    return new VectorEnumerator(this);
  }
  ...
}
class VectorEnumerator
      implements Enumeration {
  Vector vec;
  VectorEnumerator(Vector v) {
    vec = v;
  }
  ...
}
```

Figure 4: Encapsulation of a Constructor Argument. The vector `v` is encapsulated in the enumerator `e`.

4.2 Field Properties for Escape Analysis

Unfortunately, the simple analysis presented in the previous section fails to find a lot of objects that don't escape. The main idea in improving the simple analysis is to discover fields which are encapsulated, in the sense that the field is initialized by a new object by methods in the object's class or package, can only be accessed by these methods, and is never "leaked" by these methods. If we discover an object that does not escape, then the contents of any encapsulated fields of that object do not escape, and so on, recursively.

There are two conditions needed to ensure that the contents of a field never escape the containing object: (1) the value does not escape via a method that accesses the contents of the field; and (2) any value assigned to the field has not already escaped. Condition (1) is equivalent to `!may_leak`(*field*), and condition (2) is trivially true if we require that the values assigned to the field are always a newly allocated object or null. The derived property `encapsulated`(*field*) in Table 2 exactly covers these two conditions. If we have discovered an object that does not escape and a field of that object is encapsulated, then the object stored in that field also does not escape. Note that we have proved that the object in the field does not escape even though a reference to the object has been stored in the heap. For example, in the code in Figure 3, we can prove that `x` does not escape if `p` does not escape, as `x` is only stored into the encapsulated field `first` of `p`.

We also handle, via a small extension, a more general case when condition (2) is not trivially true. An object B is sometimes created in method M and then passed as a parameter to a constructor of another object A and stored in a field of A. In this case, if A does not escape and the `may_leak` property of the field is false, then B does not escape by being stored in A. A common example is the creation of an enumeration object to iterate through the contents of a list or vector. In this case, the vector is usually stored in the enumeration object, but the enumeration is used locally and does not escape. Figure 4 shows an example in Java in which an `Enumeration` object is used to enumerate the elements of a vector. The call to `elements` returns a new `VectorEnumerator` object `e` into which `v` has been stored, but `v` does not escape if `e` does not escape.

To handle this case, we define a derived property `never_leaks`, which is like `encapsulated`, but allows the source of the field to be a new object, null, *or* a parameter to a method or constructor in the field's class. If `never_leaks` is true, then the question of whether the field's contents can escape becomes equivalent to whether the input parameter can escape in the calling method. We can then use `never_leaks` to define the additional property `may_contain_param` for methods as shown in Table 4, which indicates whether a parameter is only stored into a field of the method receiver and does not otherwise leak. We also define a more precise derived property `may_store_param2`(*method*, *i*) to be false in the case that `may_contain_param`(*method*, *i*) is true.

With these new properties, we can extend our analysis in the following manner. Suppose a candidate value B is passed to a method whose receiver is the object A, and the `may_store_param2` property is false for that method, but `may_contain_param` indicates that B is stored into a field of A. Then, in determining if B escapes, we simply add the extra requirement that A must not escape either, because the only place a reference to B can be stored is into a `never_leaks` field of A.

	problem domain	**lines of code**	**Swift run-time**	
			base+ CHA	**base+ CHA+FA**
compress	text compression	910	9.78s	9.75s
jess	expert system	9734	4.11s	4.12s
cst	data structures	1800	5.79s	5.35s
db	database retrieval	1026	12.65s	12.32s
si	interpreter	1707	6.03s	5.87s
javac	Java compiler	∼18000	7.15s	7.16s
mpeg	audio decompr.	∼3600	6.41s	5.77s
richards	task queues	3637	4.78s	4.72s
mtrt	ray tracing	3952	2.01s	1.61s
jack	parser generator	∼7500	5.08s	5.03s
jlex	scanner generator	7590	4.11s	3.23s

Table 5: Java Applications

5 Performance Results

In this section, we give results showing the benefits of using field analysis. We first describe the experimental platform, the applications used in our study, and some overall performance results for field analysis. We then analyze in detail how often each field property is applicable, and how much Java overhead can be eliminated by using this field property information.

5.1 Experimental Platform

Our performance results are for the Swift compiler system running under Tru64 Unix (formerly known as Digital Unix) on an Alpha workstation. The workstation has one 667 MHz Alpha 21264 out-of-order processor, which has 64 Kbyte on-chip instruction and data caches, and a 4 Mbyte combined board-level cache. Swift is a complete Java optimizing compiler that implements numerous optimizations in addition to those described above, including: method inlining, method splitting, global common subexpression elimination, global code motion, conditional constant propagation, interprocedural alias analysis, exact type analysis, peephole optimizations, and trace scheduling. It also includes an effective register allocator based on biased graph coloring. Swift is written in Java and translates Java bytecodes to Alpha machine code. The generated code is installed into a high-performance JVM for Java 1.2 that has a mostly-copying garbage collector and extremely fast synchronization [6]. All results are for applications running with the Java 1.2 standard library.

5.2 General Results

We measure our results for a number of applications, including those in the SpecJVM98 suite. Table 5 lists the applications and problem domains, as well as the number of lines of code. Column 4 contains the running times of each application when compiled via Swift in the base configuration, with

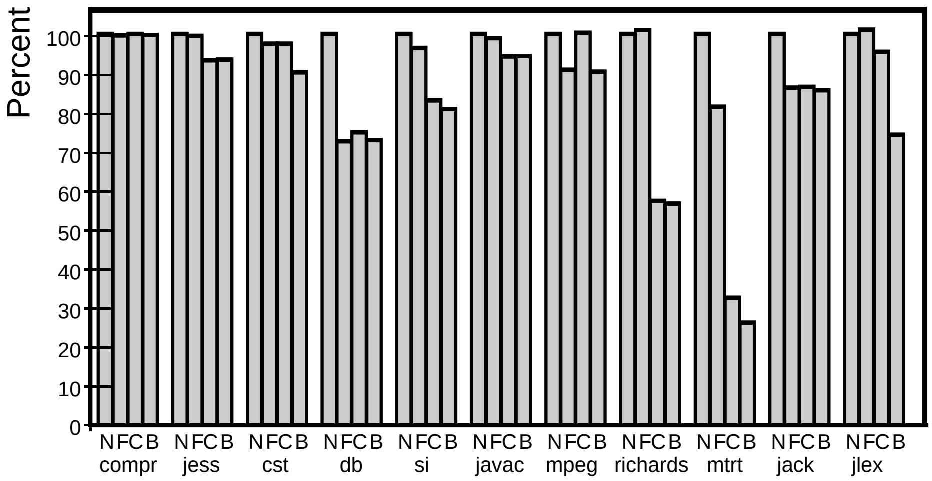

Figure 5: Timing Results

the addition of class hierarchy analysis (CHA). The base configuration includes all optimizations except those enabled by CHA and field analysis (FA). The results in column 4 are 1.18-2.33 times faster than the results for the underlying JVM. Column 5 contains the running time when field analysis and the related optimizations described in this paper are added to the results measured in column 4. The overall results show that field analysis improves performance by about 0-27%, with a geometric mean of about 7%.

Field analysis for all of these properties takes only about 10% of the compiler's processing time, which is significantly smaller than the time for register allocation. Also, we can process an average of about 15000 lines per second during field analysis. These results include the time to read the class files from the disk. The overall compilation speed is approximately the same as the speed of the platform C compiler at its highest level of optimization.

The performance improvements achieved by Swift are strongly affected by the decision to use class-hierarchy analysis. The use of CHA converts many virtual method calls to direct calls and improves the effectiveness of method inlining. In addition, its use enhances the precision of other analyses, such as alias analysis and escape analysis. However, in some situations, CHA cannot be used either because of its startup overhead, or because of the possibility of dynamically loaded classes that could invalidate the closed-world assumption made by CHA. Figure 5 shows that even without CHA, our cheap field analysis can provide significant performance improvements. The figure shows run times when using neither field analysis nor CHA (N) with the base configuration, using field analysis (F), using CHA (C), and using both (B). The times are normalized so that the base case (N) in each application is 100%.[8] The results for field analysis alone (F) find properties for `package` fields, but not `protected` fields, while the results using both (B) also find properties for `protected` fields, since all the subclasses of a class are known when CHA is used.

We observe from the graph that db, mpeg, mtrt, and jack benefit significantly from the optimizations based on field analysis, in the absence of CHA. Conversely, si, richards, and mtrt benefit strongly from CHA. Hence, field analysis is clearly useful and often helps the performance of applications in cases in which CHA does not help. There is also often synergy between the two analyses, since the combination is better than either alone in applications such as cst, mtrt, and jlex.

5.3 Detailed Results

Table 6 show the static distribution of field properties in our applications. The last column indicates the total number of reference fields in each application. The preceding columns indicate the number of reference fields with each property. These numbers are obtained for the base configuration with CHA enabled. The number of field properties discovered when CHA is disabled is slightly smaller than the number discovered when CHA is enabled. The numbers become significantly smaller in most cases if package fields are not analyzed, since some applications make minimal use of private fields.[9] Overall, the table shows that useful properties can be proved about a high percentage of the reference fields. In particular, the number of fields that are encapsulated within their containing object is quite high.

Figure 6 shows changes in dynamic counts of run-time checks. The height of the bars indicate the number of run-time checks of each kind, and all bars are normalized so that the total checks in the base case (C) is 100%. The base case includes CHA, but does not include method inlining, since inlining might change the counts in unexpected ways. Numerous run-time checks have already been eliminated in the

[8] In some cases using CHA or field analysis increases run times by a small amount. These increases are likely due to code layout changes that result in increased instruction-cache conflicts.

[9] The default access for fields without modifiers is `package`, so programmers often declare fields with `package` access, even if they could be `private`.

	exact type	non-null	const size	final	inline-able	encap-sulated	total fields
compr	5	9	8	13	2	13	22
jess	33	23	0	32	22	30	73
cst	11	5	0	6	5	16	24
db	3	3	0	4	3	1	6
si	8	5	3	6	3	9	19
javac	45	29	9	77	13	26	271
mpeg	65	103	54	109	52	70	140
richards	0	7	15	21	7	14	80
mtrt	16	11	4	27	9	12	42
jack	21	21	6	24	13	14	90
jlex	21	38	32	39	33	55	98
total	228	254	131	352	162	260	865

Table 6: Static Property Statistics for Reference Fields

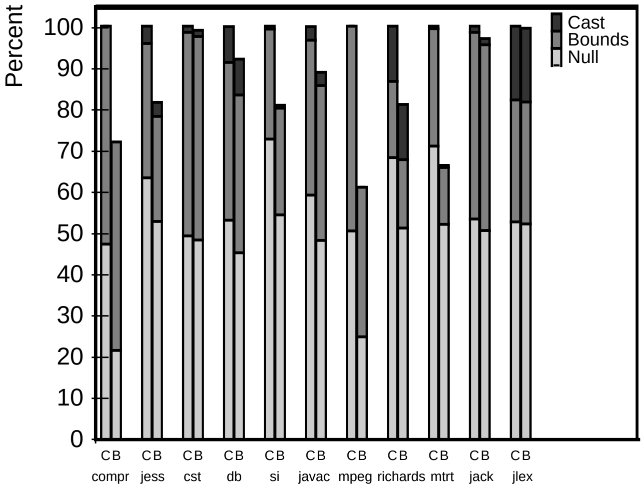

Figure 6: Dynamic Counts of Run-time Checks

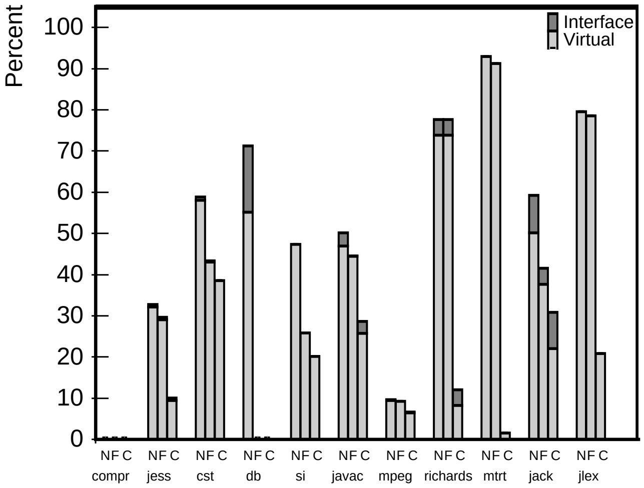

Figure 7: Dynamic Counts of Unresolved Method Calls

base case because of Swift's standard optimizations based on common-subexpression elimination, exact type analysis of values, and induction variable analysis. The second bar (B) includes both CHA and field analysis. We observe that field analysis is highly effective at removing run-time null checks and bounds checks, but not cast checks. Field analysis is also effective at removing many of the small number of array store exception checks. (These checks are included in the cast check segment, but are never more than 1% of the total checks.)

Though not shown in the graph, significant run-time checks are removed in the case when only private fields are analyzed, but the results are usually much better when package fields are also analyzed. Similarly, significant checks are also eliminated when field analysis is used without CHA, but having CHA helps in the analysis of protected fields. CHA alone eliminates only a small number of checks, mostly by increasing opportunities for common subexpression elimination.

Figure 7 shows changes in dynamic counts of virtual and interface method calls. Again, method inlining is not used. The bars are normalized so that the total number of non-static method calls (including resolved direct calls) in each applications is 100%. As before, the bar labeled 'F' represents field analysis only applied to `private` and `package` fields. CHA is highly useful for resolving method calls in most of the benchmarks. However, field analysis is nearly as effective in resolving method calls in cst, si, and jack. Also, in the results for db, javac, and jack, we see that field analysis can be useful for resolving interface calls, since it can help in determining the exact class of an object with an interface type.

As indicated in Table 6, our analysis found a significant number of fields with inlineable objects. However, the effects of inlining on performance depend, of course, on which objects are most heavily used. In our current system, we obtain a 11% improvement in mtrt because of object inlining. We expected to get an improvement in the db benchmark because of the inlining of a field in the database entry objects, but the performance improvement was only a few percent. In other applications, we found little or no improvement. We expect that there will be other applications that show significant improvement from object inlining.

Stack allocation has been fully implemented (including the case where a method M allocates an object O that leaks into one of M's callers and therefore the storage for O has to be allocated in the caller). Stack allocation results in performance improvements of 4-21% on four applications, and minimal improvements for other applications. We have not included more detailed results in this paper, because, even though field analysis can help identify extra opportunities for stack allocation, in our current benchmarks it does not actually identify any new opportunities (but it does identify more opportunities for synchronization removal, as described next). Our current implementation will allocate `String` and `StringBuffer` objects on the stack, but does not allow stack allocation of variable-sized arrays, so stack allocation of the arrays inside `String` and `StringBuffer` objects is not currently possible.

Swift also implements synchronization removal. For each object which does not escape a thread, Swift descends the

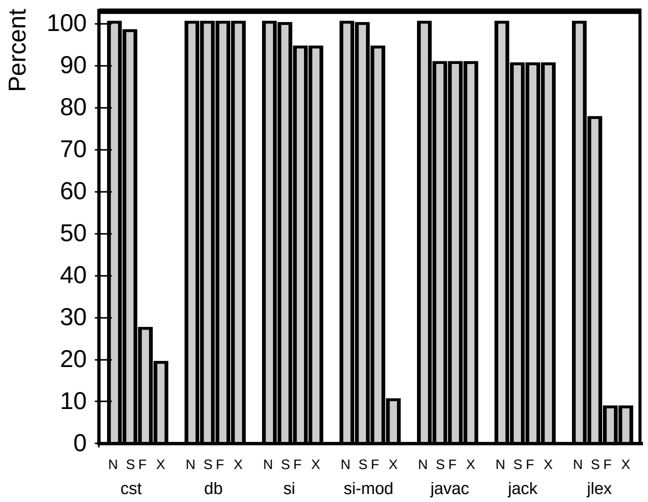

Figure 8: Dynamic Synchronization Counts

call chain of methods that take that object as an argument and compiles special unsynchronized versions of the methods where necessary. It may also have to compile alternate versions of any intermediate methods in the call chain, even if they don't synchronize. Swift also removes synchronization from any encapsulated fields of the object during the same traversal. Figure 8 shows the percentage of synchronizations that are eliminated by our optimizations for the applications that actually do synchronization. For each application, the first bar ('N') represents the total number of synchronizations with no optimizations, the second bar ('S') represents the number when the simple analysis of Section 4.1 is applied, the third bar ('F') represents the number when the additional information about encapsulated fields is used, and the last bar ('X') represents the number when the extension in Section 4.2 is used.

Our techniques are sometimes foiled by complicated Java library code, however. A large percentage of the synchronization in javac and si is on encapsulated file stream handles that don't escape, but our analysis was defeated by the many subclasses of the standard file stream classes, some of which have unusual effects on the common fields. To handle this case, we would need to do extensive context-sensitive analysis and cloning or change the applications to use a less complex file stream hierarchy. As an experiment, we modified si to use a private version of `PushbackInputStream` that is modified so it does not inherit from `FilterInputStream`, which is the source of many of the analysis problems. The results for the modified version of si (which has no other changes) are labeled as 'si-mod'.

Figure 8 shows that the simple analysis eliminates 5-20% of the synchronizations in most of the applications. However, the use of encapsulated field information dramatically improves the success of the analysis for cst, si-mod, and jlex. In addition, the extension described in Section 4.2 is crucial for eliminating the synchronizations in si-mod. The reason is that most synchronizations occur on a `ByteArrayInputStream` object, which is first created, and then stored inside a `PushbackInputStream` object. Hence, our extension is important for handling the common structure of the multilayer input streams created using the Java library. Overall, our synchronization removal improves the performance of cst, si-mod, and jlex by 8-27%.

6 Related Work

There has been a variety of recent work that attempts to use exact type or field information to improve the efficiency or accuracy of some kind of analysis. Type-based alias analysis [10] makes use of the strong typing properties of languages like Java to eliminate potential aliases between references because they access different fields or base types. Ruf [17] has used type information to partition dataflow problems for efficiency. However, we are unaware of any such type- or field-based analysis that attempts to prove properties about the contents of fields.

Diwan *et al.* [11] proposes the use of *aggregate analysis* to detect when a polymorphic data structure is used in a monomorphic way by looking at the types of all assignments to a particular field. For example, the analysis is used to show that a linked list of general objects actually contains only objects of a certain class or its subclasses. This analysis is related to our field analysis that determines exact types, but aggregate analysis does not make use of any modularity properties in the language, or investigate any other properties of fields. Detlefs and Agesen [8] describe a use for the `final` property of fields, which they call "immutability", when deciding whether to inline a resolved virtual method call in the presence of dynamic loading. For their application, they compute immutability for `private` fields only, and do not appear to detect the case when `this` might be "leaked" in a constructor.

Program checkers such as ESC-Java [9] allow annotations that specify properties of fields that are verified as part of the checking process. These properties include a specification that a field is non-null or that a method returns a new, unaliased object. Our compiler attempts to discover these properties automatically, rather than using programmer annotations.

Dolby and Chien [12] describe an object inlining optimization for C++ programs. Their analysis uses a fully context-sensitive interprocedural framework and thus allows object inlining in specific cases on a field that cannot be inlined in general. In particular, they do not require that the field be initialized with a new object in the constructor. However, their analysis times are measured in minutes, whereas our analysis is always only a small number of seconds. Also, we allow objects to be inlined (with a header), even if a reference to the objects escape the local context. A large amount of related work with respect to object inlining (or *unboxing*) also exists for functional languages [14, 16, 19], as described in [12].

There have been several recent descriptions of optimizations based on escape analysis. Most of these have focused on converting heap allocations to stack allocations and removing unnecessary synchronization. Blanchet [3] describes a technique where type information is summarized by an integer height. Interprocedural analysis is then used to compute what is leaked by each method. The approximation of types by integers leads to some imprecision, but in experiments, this lack of precision has not made a difference. The analysis itself is quite fast and effective. Bogda and Holzle [4] use a two-pass interprocedural analysis to remove unnecessary synchronization. Their scheme extends the simple analysis of Section 4.1 to one more level (although it is flow-insensitive), and can prove that an object O does not escape if it is only reachable via a path containing one heap object (not including itself). The performance results are fairly promising, but the paper has no data on the efficiency of the analysis itself. Our approach also extends the simple analysis, but can potentially prove objects don't escape even if reachable by a path of multiple heap objects.

Whaley and Rinard [21] and Choi *et. al.* [5] perform escape analysis by using a variation on the *points-to* graph generally used in alias analysis. The precision of these schemes can by tuned by changing the context sensitivity of the analyses. Aldrich [2] finds all groups of objects that might be accessible via a global variable (and hence escape a thread). The analysis requires the entire program, since it proceeds by marking sets of objects that might be stored in global variables, then sets of objects that might be stored in these objects, and so on, until a fixed point is reached. Only Blanchet [3] reports on analysis efficiency, so it is hard to compare most of these systems with our system in terms of compilation time.

As far as we know, none of the preceding systems use field access properties to aid in their escape analysis. Some Java compilers no doubt perform extra optimizations for `final` fields, but none seem to have exploited field properties to the same extent as Swift.

7 Conclusion

We have described field analysis, an interprocedural analysis that determines useful properties of fields in a low-cost manner by exploiting the modularity properties of a language. Field analysis does not require analysis of an entire program and can eliminate numerous run-time checks that are not eliminated by other optimizations. In addition, it can also be effective in resolving virtual method calls, and is useful in situations where class hierarchy analysis is not appropriate. Field analysis takes only about 10% of the compile time and is inexpensive enough to be useful in a compiler that is invoked dynamically during the execution of an application.

We have also shown that field analysis can help reduce the analysis burden of optimizations such as object inlining, stack allocation, and synchronization removal. By making use of properties discovered via field analysis, we have implemented useful versions of these optimizations that are much less inexpensive than corresponding versions which do full interprocedural dataflow analysis. Overall, speedups for all the optimizations enabled by field analysis average 7% on a set of programs, including the SpecJVM98 applications, with some application times reduced by up to 27%.

There are many other field properties that might be useful to investigate. If analysis shows that an integer field is always assigned values within a certain range, then the storage size for the field can potentially be reduced by the compiler. It would also be useful to prove that elements of an array referenced by a field are always assigned objects of a specific class. Finally, even more complex properties may be useful: A common idiom in object-oriented programs is to have a class that contains an array and also a "current" index into the array. With field analysis, the compiler may be able to prove that the index field is always less than the length of the array. If so, the compiler will be able to eliminate bounds checks whenever the index is used to load an element from the array.

Acknowledgments

We would like to thank Jeff Dean, Raymie Stata, and Mark Vandevoorde, who worked on the early design and implementation of Swift, and Girish Kumar, who worked on the first implementation of stack allocation. We would also like to thank Amer Diwan and the anonymous referees for their comments.

References

[1] Java Memory Model Mailing List. At URL `http://www.cs.umd.edu/~pugh/java/memoryModel/`.

[2] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Sixth International Static Analysis Symposium*, Sept. 1999.

[3] B. Blanchet. Escape Analysis for Object Oriented Languages. Application to Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.

[4] J. Bogda and U. Hölzle. Removing Unnecessary Synchronization in Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.

[6] Compaq Computer Corporation. Compaq Fast Virtual Machine V1.2.2-1 for Alpha. At URL `http://www.compaq.com/java`.

[7] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP '95*, pages 77–101, Aug. 1995.

[8] D. Detlefs and O. Agesen. Inlining of Virtual Methods. In *ECOOP '99*, pages 258–278, June 1999.

[9] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report 159, Compaq, 1998.

[10] A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-Based Alias Analysis. In *1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1998.

[11] A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and Effective Analysis of Statically-Typed Object-Oriented Programs. In *1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1996.

[12] J. Dolby and A. Chien. An Evaluation of Automatic Object Inline Allocation Techniques. In *1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1998.

[13] D. Gay and B. Steensgaard. Stack Allocating Objects in Java. At URL `http://www.research.microsoft.com/apl/stackalloc-abstract.ps`.

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

[15] T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Prog. Lang. Syst.*, 1(1):121–141, July 1979.

[16] X. Leroy. Unboxed Objects and Polymorphic Typing. In *19th Symposium on the Principles of Programming Languages*, pages 177–188, Jan. 1992.

[17] E. Ruf. Partitioning Dataflow Analyses Using Types. In *ACM SIGPLAN/SIGACT '97 Symposium on Principles of Programming Languages*, Nov. 1997.

[18] D. J. Scales, K. H. Randall, S. Ghemawat, and J. Dean. The Swift Java Compiler: Design and Implementation. Technical Report 2000/2, Compaq Western Research Laboratory, Apr. 2000.

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

[20] V. C. Sreedhar and G. R. Gao. A Linear Time Algorithm for Placing Phi-nodes. In *22nd Annual ACM Symposium on Principles of Programming Languages*, Jan. 1995.

[21] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.