A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading

Vugranam C. Sreedhar Michael Burke Jong-Deok Choi
IBM T. J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598
{sreedhar,burkem,jdchoi}@watson.ibm.com

ABSTRACT

Dynamic class loading during program execution in the Java^{TM} Programming Language is an impediment for generating code that is as efficient as code generated using static whole-program analysis and optimization. Whole-program analysis and optimization is possible for languages, such as C++, that do not allow new classes and/or methods to be loaded during program execution. One solution for performing whole-program analysis and avoiding incorrect execution after a new class is loaded is to invalidate and recompile affected methods. Runtime invalidation and recompilation mechanisms can be expensive in both space and time, and, therefore, generally restrict optimization.

To address these drawbacks, we propose a new framework, called the *extant analysis framework*, for interprocedural optimization of programs that support dynamic class (or method) loading. Given a set of classes comprising the *closed world*, we perform an offline static analysis which partitions references into two categories: (1) *unconditionally extant* references which point only to objects whose runtime type is guaranteed to be in the closed world; and (2) *conditionally extant* references which point to objects whose runtime type is not guaranteed to be in the closed world. Optimizations solely dependent on the first category can be statically performed, and are guaranteed to be correct even with any future class/method loading. Optimizations dependent on the second category are guarded by dynamic tests, called *extant safety tests*, for correct execution behavior. We describe the properties for extant safety tests, and provide algorithms for their generation and placement.

1. INTRODUCTION

Dynamic class loading during program execution in the Java^{TM} Programming Language [18; 25] is an impediment for generating code that is as efficient as code generated using static whole-program analysis and optimization. Sophisticated static compilers for languages such as C++ perform whole program analysis, optimization, and transformation, to generate efficient code. Whole program analysis is possible for them since they do not allow new classes/methods to be loaded during program execution.[1]

In Java new classes can be loaded on-the-fly during program execution [25]. Attempting to apply whole program static analysis framework to Java can result in an incorrect program. For instance consider a virtual call $p.foo()$. In C++, using whole program analysis we can determine whether a virtual call has only one target [3]. If so, the virtual call can be directly converted to a static call (and the call possibly inlined). Attempting to do such devirtualization in Java (without a runtime type check guarding the devirtualization) can result in an incorrect program because, during execution, a new class can be loaded and $p.foo()$ can invoke a new $foo()$ in the newly loaded class. One solution for avoiding incorrect execution after a new class is loaded is to invalidate and recompile affected methods [5; 6; 19; 21]. Runtime invalidation and recompilation mechanisms have several drawbacks: (1) they can be expensive in both space and time; (2) the activation stack frame for active and invalidated methods may have to be rewritten [19]; and (3) they can restrict how much optimization one is allowed to do so that invalidation can be correctly applied during runtime [6; 19]; and (4) a complex and an expensive synchronization mechanism may be needed to correctly invalidate methods in a multithreaded environment [2].

To address these drawbacks, we propose a new framework, called the *extant analysis framework*, for interprocedural optimization of programs that support dynamic class/method loading. Given a set of classes comprising the *closed world*, we perform an offline static interprocedural analysis as if this set made up a whole program. In addition, the offline analysis performs an *extant analysis* which partitions references into two categories: (1) *unconditionally extant* references which point only to objects whose runtime type is guaranteed to be in the closed world; and (2) *conditionally extant* references which point to objects whose runtime type is not guaranteed to be in the closed world.

In this framework we perform unconditional static optimizations (such as direct inlining) for the first category. For the second category we identify regions of code (methods or parts of methods) for which it is beneficial to optimize

[1] In the presence of calls to methods whose body is not known during compile time, such as Dynamic Link Libraries (DLLs), these static compilers usually make conservative assumptions about the methods.

PLDI 2000, Vancouver, British Columbia, Canada.

```
Class A{
   public static void foo(C c)
   {
       ...
       B b = goo() ; // b1
       b.Bbar() ;  // cs1
       c.Bbar() ;  // cs2
   }

   static B goo() { return new B(); // cs3 }

   public void bar(C c1)
   {
       foo(c1);  // cs4
   }
}
```

```
class B {
    public void  Bbar ()
    {
       ...
    }
}

class C extends B {

}
```

Figure 1: An example program.

and create a specialized version of the optimized code. We generate code called the *extant safety test* to be executed during runtime for a safe invocation of the specialized code. Within our framework we *guarantee* that once specialized code is invoked, it will execute correctly and safely for *that invocation* even when new classes are loaded.

We now illustrate our framework using the example of Figure 1. We use the notation C::m to mean that the method m is defined in class C. Classes `A`, `B`, and `C` are the only classes available for off-line analysis. We refer to classes available for the offline analysis as "extant classes," and an object whose runtime type is an extant class as an "extant object." With respect to the "closed-world" program of the extant classes, "extant analysis" determines that the class of the object pointed to by `b` at `cs1` is guaranteed to be within the closed world: the class (`B`) of the object created at `cs3`. A data flow analysis with respect to the closed-world program determines that the method invocation `b.Bbar()` at `cs1` has only one target, i.e., `B::Bbar()`. For any invocation of `foo(C c)`, therefore, `b.Bbar()` at `cs1` can be devirtualized and inlined.

Extant analysis also determines that the object pointed to by `c` at `cs2` is the same as that pointed to by parameter `c` at the entry of `foo(C c)`. `A::foo(C c)`, being a public method, can be invoked from outside the closed world, and the object pointed to by `c` at `cs2` can be extant or non-extant. A data flow analysis with respect to the closed-world program determines that if `c` at `cs2` points to an extant object, the method invocation `c.Bbar()` at `cs2` has only one target, `B::Bbar()`. Whereas `b.Bbar()` at `cs1` can be directly inlined, a runtime test is needed to guard the inlined `B::Bbar()` invoked at `cs2`.

Since `c` does not change within `A::foo(C c)`, we can further specialize the entire body of `foo(C c)` with respect to the assumption that the runtime type of `c` is extant. We designate the specialized version of `foo(C c)` as `foo´(C c)`. This specialized version can be safely invoked by any call site invoking the method that passes an extant object as the parameter. Since `c1` at `cs4` is passed as the parameter to `A::foo(C c)`, we can place a runtime "extant safety test" prior to `cs4`. At runtime, this test determines whether `c1` points to an extant object. If so, the specialized `foo´(C c)` is invoked; if not, the original `foo(C c)` is invoked. The transformed code at `A::bar()` looks like

```
if (c1 points to  an extant object)
      foo´(C c);
  else
      foo(C c);
```

Extant analysis statically determines the set of call sites such as `cs1` where the target method is guaranteed to be in the closed world, and call sites such as `cs2` where the target method can be within or outside of the closed world. In the extant analysis framework, we also determine program locations (such as `cs4` for call site `cs2`) for placing a runtime extant safety test that can maximize optimization opportunities, and determine how to generate and perform such a test.

Detlefs and Agesen [15] introduce the concept of *preexistence* in the context of a *dynamic compiler*. Here inlining only takes place for those call sites for which the object pointed to by the receiver expression has already been allocated at the moment of invocation of the containing method. Using their preexistence technique, the method invocation at `cs1` cannot be inlined, because the object pointed to by `b` is not allocated until after the invocation of `foo()`.

Another approach to devirtualization and inlining is based on a runtime type check of the receiver expression [20]. Using this technique, one can devirtualize and inline `c.Bbar()` at `cs2` as follows:

```
if (c instanceof B || c instance of C) {
  // devirtualize and inline B::Bbar()
}
else
{
    c.Bbar();
}
```

In contrast to the runtime type check, our extant safety test can cover the entire specialized method. This offers the opportunity for optimizations across the multiple statements of the specialized method. Further, the statements covered by a single extant test can cross method or class boundaries, in which case interprocedural optimizations can be performed across muliple levels of method invocations.

In this paper we do not address the Java features of reloading and the Java Native Interface.[2]
We implemented the framework using the Jalapeño Virtual Machine [1] and experimented on a set of SPECjvm98 benchmark programs and on **portBob** [4]. We present our preliminary experimental result in Section 6.
The main contributions in this paper are as follows:

- We introduce a framework for interprocedural optimization that does not require invalidation during execution when a new class is loaded.
- We introduce an analysis technique, called *extant analysis*, to identify code regions that can be specialized and to identify points in the program that can be affected by dynamic class loading.
- We use *parametric data flow analysis* as a basis for performing optimizations on the specialized code regions.
- We introduce an *extant safety test* that can be performed during execution to safely invoke the specialized code regions. We show how to generate and perform such tests. We also formalize the safety properties of an extant safety test.
- We experimented with our framework using Jalapeño Virtual Machine. We present both static and dynamic results to evaluate our framework.

The rest of the paper is organized as follows: Section 2 describes the relationship between Java dynamic class loading and the framework for offline analysis/ optimization used in this paper. Section 3 describes extant analysis. Section 4 describes our techniques for identifying candidate code regions for specialization and generating specialized code regions. Section 5 describes the placement and generation of extant safety tests. Section 6 describes our implementation and experimental results. Section 7 discusses related work, and Section 8 gives our conclusion.

2. DYNAMIC CLASS LOADING AND THE CLOSED WORLD

In Java there are several ways in which a new class can be loaded. According to the Java specification, during execution a new class should be loaded if there is a reference to an element (such fields or methods) of the new class [18; 26]. There are two other ways of loading a new class in Java: (1) via `Class.forName()` constructs and (2) via user defined class loaders [25].[3]
When Java is implemented in a static environment, such as JAX, IBM HPCJ, NaturalBridge, or Marmot, the static compiler expects the whole program to be present during the analysis [22; 17; 28; 33] HPCJ, for instance, searches all methods and classes that are reachable from the `main()` method in the main application class. But when it encounters a `Class.forName()` or user defined class loaders, it relies on the user to provide all possible classes that can be loaded at such points. If a user fails to provide all classes that can be dynamically loaded at such points, then HPCJ would potentially generate incorrect code.
In our framework we also rely on the user to provide methods and classes that can participate in the offline analysis and optimization phase. But unlike a static Java environment, our framework guarantees correctness even if the user does not provide all of the dynamically loadable classes. We call the set of classes and methods that participate in *offline* analysis and optimization, the *closed-world program*. There are several ways to construct the closed-world program: (1) use the same strategy as that of a static Java compiler, such as HPCJ, except that the user need not specify all possible classes that can be loaded at `Class.forName()` points or by user-defined class loaders; (2) use profiling information and include only hot methods and classes; and (3) the user can specify the set of all classes and methods that participate in the closed world.

[2]These Java features could potentially modify code on-the-fly and so could affect our assumptions about the closed world.
[3]Class loaders allow the same fully qualified classes to have different runtime types during execution. A runtime type of an object is a pair $\langle L, C\rangle$, where L the defining class loader for C. The offline analysis phase must be aware of class loaders to ensure type safety of the program [25; 31].

3. EXTANT ANALYSIS

In this section we describe *extant analysis*. First we define certain useful terms.
Class inheritance relationships in Java can be represented using a Class Inheritance Graph (CIG) $G = (N, E, r)$ where N is a set of nodes representing either a class or an interface, E is a set of edges representing inheritance relation, and r is a distinguished root node representing the class `java.lang.Object`. We will sometimes use the term *type* to mean either a class or an interface, *sub-type* to mean subclass or sub-interface. An edge $(x, y) \in E$ represents that x is an immediate super-type (also called the parent type) of y, and y is an immediate sub-type (also called the child type) of x. If there is a path from a type x to another type y in G, then x is called the ancestor type of y, and y is called the descendent type of x.
Given a declaration of the form "`T p`", `T` is called the *declared type* of the reference variable `p`. During program execution the type `R` of the object that `p` points to can be any class that is (directly or indirectly) derived from `T`. We will call `R` the *runtime type* or *concrete type* of the object that `p` is pointing to. Now let `p` be a reference (to an object) and `p.foo()` a method call. We call `p` in `p.foo()` the *receiver expression*, and the type of the receiver expression is the runtime type of `p`. We will use the term virtual call to include both `invokevirtual` and `invokeinterface` calls.

DEFINITION 3.1 (CLOSED-WORLD SET). *The set of classes and methods that participate in off-line analysis and optimization is called the* Closed-World *set, and is denoted by* $\mathcal{CW}$.

DEFINITION 3.2 (CONNECTED CLOSED-WORLD SET). *Let x and y be any two classes in a $\mathcal{CW}$ such that x is an ancestor of y in the CIG. Let P be a set of all paths between x and y. $\mathcal{CW}$ is said to be a* connected closed-world *if and only if all types in P are also in $\mathcal{CW}$. $\mathcal{CW}$ is said to be a* root-connected closed-world *if x is the class* `java.lang.Object` *(i.e., the root of the CIG).*

A closed world's having the "connected" property simplifies the analysis and aids in program optimization. Through-

```
public void m1(String cName)
{
     B p;
     if (...)
S1:    p = new B();                          // extant
     else
     {
S2:    Class c = Class.forName(cName);
S3:    Object o = c.newInstance();
S4:    p = (B) o;               // non-extant
     }
S5: for (...)
     {
S6:   p.Bbar(...);      // p is extant or non-extant
     }
}
```

Figure 2: Example for Extant Analysis

out this paper we will assume a *root-connected* closed world, unless otherwise stated.

DEFINITION 3.3. *A class C (or a method M) is* extant *if and only if C (or M) is in $\mathcal{CW}$. An object O is* extant *if and only if its runtime type is extant.*

We will sometimes state that a class or method is not extant through use of the term non-extant.

DEFINITION 3.4. *A reference (such as a receiver expression) is* unconditionally extant *at a program point (call site) if and only if it can only point to an extant object at that program point (call site); otherwise it is* conditionally extant.

Since we use the terms class and type interchangeably we sometimes say that a "type is extant" instead of saying that a "class is extant."

DEFINITION 3.5. *A virtual call site is unconditionally extant if and only if the receiver expression at that call site is unconditionally extant (otherwise it is* conditionally extant*). A static call site is extant if and only if the target method of the call site is in $\mathcal{CW}$.*

Extant analysis is a data flow problem for computing the "extant state" of the receiver expression at each virtual call site. Extant analysis is performed during the offline analysis phase. The lattice for extant analysis consists of three elements: UNKNOWN ($\top$), *unconditionally extant* (UCE), and *conditionally extant* (CE or $\bot$). Let $ES = \{\top, UCE, \bot\}$ be the set of lattice elements (extant states). Let $A \in ES$, then $\bot \wedge A = \bot$, and $\top \wedge A = A$. If a receiver expression is conditionally extant, we can then establish a "degree of extantness". For instance, if we can determine statically that a receiver expression can never point to an extant object, then we say it is *unconditionally non-extant* (UCNE).[4] We will use the concept of "degree of extantness" in Section 4 for specialization.

We compute the extant state of the receiver expression of each invocation site in two steps:

1. The extant state of the receiver expression is set to $\bot$ if the target method is not implemented in the declared type D of the receiver expression, or in any ancestor or descendant of D in $\mathcal{CW}$.

2. For any other receiver expressions, apply the *meet* ($\wedge$) operation to the extant states of the set of *compile-time objects* the receiver expression can point to.

If the set of compile-time objects of the receiver expression is empty, the extant state of the receiver expression is set to $\bot$ for safety in optimization. This can happen for a receiver expression reached by a parameter of a non-public method in the closed world.

The set of compile-time objects a reference variable can point to can be computed by applying a pointer analysis algorithm [16; 34; 24; 9; 32; 30]. A compile-time naming scheme similar to those used in pointer analysis identifies compile-time objects [7; 30].

The extant state of a compile-time object is initialized with UCE (unconditionally extant) if the object is allocated with a type that is in the closed world. Otherwise, its extant state is initialized with $\bot$.[5]

In Java a public method can be invoked from outside the closed world. Let $M(\ldots)$ be a method that can be invoked from outside the closed world. If p is a reference parameter of M, then p can also potentially point to a non-extant object. To reflect this, when a compile-time object is passed as a parameter to a public method, we apply a *meet* with $\bot$ to the extant state of the compile-time object. If M is not reachable from any method in the closed world, each reference parameter of M is assigned a compile-type object with extant state of $\bot$. We call methods such as $M(\ldots)$ *entry boundary points* to the closed world.

When a reference variable is passed as a parameter to a method invocation on a potentially non-extant object, the extant state of the objects indirectly reachable from the parameter can become non-extant because the objects can be replaced. We apply a *meet* with $\bot$ to the extant state of objects indirectly reachable from parameters passed to a method invocation on a potentially non-extant object. Reachability information is implicit in the pointer analysis solution.

A statement which performs a read of a static reference variable is also an entry boundary point, as such a variable may have been modified by another thread during the execution of the containing method.[6] Therefore, when a compile-time object becomes reachable from a static reference variable, we also apply a *meet* with $\bot$ to the extant state of the compile-time object.

Consider the example shown in Figure 2. At `S1` the reference variable `p` points to an extant object (assuming that `B` is extant), whereas at `S4`, `p` can potentially point to a non-extant object (assuming that the value of `cName` is not

[4] Sometimes it is convenient to assume a four element lattice $\{\top, UCE, UCNE, \bot\}$, with $UCE \wedge UCNE = \bot$. For instance, if we know that a reference will always point to a non-extant object, then we can set its extant state to $UCNE$. As we will discuss in Section 4, if the receiver expression of a virtual call site has extant state UCNE, then it is not beneficial to optimize with respect to such call sites.

[5] Its extant state is initialized with UCNE in the four element extant-state lattice.

[6] We cannot eliminate static variables from consideration by using techniques similar to "lambda-lifting" for free variables, due to multiple threads.

known during off-line phase). Therefore at S6, the receiver expression p can invoke a method Bbar(...) that may not be in the closed world.

PROPERTY 3.1. *If a receiver expression p at a call site p.foo() is unconditionally extant, then during execution all possible target methods of the call are within the closed world.*

An important implication of the above property is that we never need a runtime type check with respect to an optimization of such unconditionally extant call sites. For instance, we can directly devirtualize and inline such call sites without any runtime type checks if, during offline analysis, we can determine that there is only one target for such virtual calls.
If a receiver expression is conditionally extant, then the call site can potentially invoke methods outside the closed world. We call such call sites *exit boundary points* of the closed world. The invocation of p.Bbar(...) at S6 in Figure 2 is an example of an exit boundary point. We need to specialize code regions containing optimizations depending on such call sites, where the invocation of the specialized code is guarded by a runtime type check.
In the next two sections we will use extant analysis information (1) to compute code regions that are candidates for specialization; and (2) to compute the extant safety test for determining whether to call the specialized or unspecialized code during execution.

4. CODE SPECIALIZATION

In this section, using extant analysis, we will first show how to identify methods or parts of methods as candidates for specialization (Section 4.1). We then show how to use concepts from parametric data flow analysis to generate specialized code. (Section 4.2).

4.1 Identifying Specialization

In our framework a method or a part of a method is always specialized with respect to an exit boundary point in the closed world. In the previous section we used extant analysis to determine the extant state of a receiver expression. If the extant state of the receiver expression p at a virtual call site $p.foo()$ is UCE, then we can perform unconditional optimization with respect to such call sites. But if the extant state of p is $\perp$ we can do one of the following (depending on the "degree of extantness"): (1) do not perform any optimization with respect to such exit boundary points; or (2) perform optimizations with respect to such exit boundary points but guard the optimized code using a dynamic test. In the remainder of this section we describe how to identify code regions that are candidates for optimization. We optimize code with respect to an exit boundary points only if it is beneficial to do so. Therefore in the remainder of this section we assume that the receiver expression is *not* unconditionally non-extant.
To motivate the problem, consider the simple example shown in Figure 3. Assume that all classes shown in the figure belong to the closed world. The compile-time object created at S10 and passed to method A::Abar($\cdots$) as the first parameter, i.e. c1, has its initial extant state as UCE. Since A::Abar($\cdots$) is a public method, the extant state of the compile-time object is applied to a *meet* with $\perp$, resulting in the extant state of CE. c1 is passed to parameter c of A::Afoo(C c) at S1. The extant state of receiver expression c at S5 does not change in A::Afoo(C c), and we apply specialization to A::Afoo(C c) to create a specialized A::Afoo´(C c) in which we *directly* devirtualize and inline c.Bbar(b) at S5 (without any runtime guards controlling the inlined method). A::Afoo(C c) is specialized here with respect to the exit boundary point defined at S5. During execution we perform an *Extant Safety Test (EST)* at S1 to determine whether to call the specialized version A::Afoo´(C c) or to call the unspecialized version A::Afoo(C c). Once A::Afoo´(C c) is invoked, it can safely execute for that invocation even if a new class is loaded that can override the method B::Bbar(B b).
Now consider the method A::Agoo(C c, String s). We cannot specialize the whole method A::Agoo(...) wherein we can directly inline c.Bbar(b). In both A::Afoo(C c) and in A::Agoo(...), the receiver expression c for invocations of c.Bbar(b) at S5 and S9, respectively, can point to a non-extant object. But in the case of A::Afoo(C c) a new class can be loaded only prior to the execution of S1, whereas in the case of A::Agoo(...) a new class can be loaded at S7, and the receiver expression at S9 can point to the newly loaded class. We can specialize part of A::Agoo(...) by placing an extant test just prior to S9 to check whether the receiver expression can point to a non-extant object generated at S8. In the next section we will show how to generate and place extant tests.
Next we establish two key properties to determine which methods (or parts of methods) are candidates for safe specialization. Intuitively, given an exit boundary points χ, we can place an EST just prior to χ. But we want to move this test as early as possible to create opportunities for other optimizations. The next property essentially states how far up can the test be moved from an exit boundary points.

PROPERTY 4.1. *Let m be a program point, let χ be an exit boundary point, and let $\mathcal{P}(m,\chi)$ be the set of all program paths from m to χ in the program. The statements in $\mathcal{P}(m,\chi)$ can be* safely *specialized with respect to χ if, during execution, (1) there does not exist a non-extant (runtime) object O_{ne} that can reach χ before reaching m and (2) any object that reaches both m and χ is extant.*

In Property 4.1, if m is an entry point to a method M, then we can specialize the whole method M with respect to χ. The portions of M that do not lie on a path from m to χ will not affect the safety of the specialization. Let us dissect Property 4.1 into three parts: (1) program point m, (2) the set of paths $\mathcal{P}(m,\chi)$, and (3) the exit boundary point χ. At compile-time we want to identify m such that $\mathcal{P}(m,\chi)$ can be safely optimized and specialized with respect to χ. To invoke the specialized code we want to place an EST EST_m at m that ensures safe invocation of the specialized code. For an invocation of a path P in $\mathcal{P}(m,\chi)$ to be safe, we have to ensure that whenever P is executed, the receiver expression re_χ at χ points to an extant object. Let p_χ be the receiver expression at χ, and $EST(p_\chi)$ be an EST performed at χ that returns true if and only if p_χ points to an extant object. Any EST EST_m that we place at a program point m should satisfy the following property:

PROPERTY 4.2. *An EST at program point m, denoted as EST_m, is* safe *with respect to $EST(p_\chi)$ if EST_m implies $EST(p_\chi)$, expressed as $EST_m \sqsubseteq EST(p_\chi)$.*

```
Class A{
   public void  Abar(C c1, String s)
   {
  S1:  Afoo(c1);
  S2:  Agoo(c1, s);
   }

   private void Afoo(C c)
   {
  S4:  B b = new B();  // extant
  S5:  c.Bbar(b);
   }

   private void Agoo(C c, String s)
   {
       ...
  S6:  if(...) {
  S7:   Class x = Class.forName(s)
  S8:   Object c = x.newInstance(); // non-extant
       }
  S9:  c.Bbar(b);
       ...
   }
}
```

```
class B {
    public void  Bbar (B b)
    {
       ...
    }
}

class C extends B {
   ...
   public void Efoo()
   {
     ...
   }
}

class D {
   ...
   public void Dbar(A a1)
   {
 S10: C cc = new C();
 S11:  a1.Abar(cc, "...");
   }
}
```

Figure 3: A closed-world program.

In other words, whenever EST_m is true, $EST(p_\chi)$ should be true. But if EST_m is false, $EST(p_\chi)$ can be either true or false. We call EST_m a *surrogate* of $EST(p_\chi)$. We say EST_m is stronger than EST_n (or EST_n is weaker than EST_m) if $EST_n \sqsubseteq EST_m$, where EST_m and EST_n are two surrogates (possibly at the same program point) of $EST(p_\chi)$.

Consider the example program in Figure 3. The set of objects pointed to by the receiver expression `c` at `S5` is the same as the set of objects pointed to by `c` at the entry of `A::Afoo()`. But this is not true in the case of `A::Agoo()`. Therefore `A::Agoo()` cannot be fully specialized. But if we place an EST immediately prior to `S9`, the part of the program from the EST point to the exit boundary point can be specialized. In Section 5, we show how to compute the EST.

4.2 Specialization Using Parametric Data Flow Analysis

In the previous section we illustrated how to specialize methods for a devirtualization optimization. The optimization was based on a closed-world analysis performed with respect to an exit boundary point. A closed-world analysis which is parameterized with respect to exit boundary points is an example of a parametric data flow analysis [29; 8]. For devirtualization, the candidate method (or partial method) for specialization contains the exit boundary point. There are optimizations, such as stack allocation based on escape analysis [11], where the method to be specialized does not contain the exit point on which it depends. For such problems an interprocedural *parametric* data flow analysis is needed to generate the specialized methods.

Parametric data flow analysis is based on augmenting data flow analysis with the computation of conditions on which the validity of the analysis depends. Let $\langle c, f \rangle$ denote a data flow fact f whose truth value depends on the condition c. Suppose T is a transformation that uses this fact. Then the transformation T is correct if the condition c holds during program execution. Once the condition c becomes false, the transformation T is unsafe.

Consider escape analysis for compile-time garbage collection. Let `p = new R()` be an allocation site in a method M that we wish to transform to `p = newstack T()`. This transformation T is safe only if we can prove that the objects allocated at this site cannot escape M. Let O be the compile-time object name for this site. Using parametric escape analysis we can compute parameterized escape information $\langle \Xi, O \rangle$, where Ξ is the set of exit boundary points that could potentially affect the escapement of O. Assume O does not escape M under the condition that none of exit boundary points in Ξ will target a method outside the closed world. Now we can specialize T with respect to all exit boundary points in Ξ, and place an EST guarding the specialized transformation. We specialize T only if we can find a placement for an EST that will satisfy Property 4.2.

5. EXTANT SAFETY TEST

Extant Safety Tests (ESTs) are condition-checks that guard the safe execution of a specialized method (or portions of a method). These tests depend on runtime information. There are several ways to perform the EST. Any test that we perform, however, must satisfy Property 4.2. Consider Figure 3. We can place the following EST at `S4`:

```
S4:  if(EST(c1)) Afoo´(c1) /* specialized */
     else  Afoo(c1) /* unspecialized */
```

`EST(c1)` returns true if `c1` points to an extant object, otherwise it returns false. `EST(c1)` is a surrogate for `EST(c)` at S2. ESTs that query objects for safety can be implemented

by adding a bit in the *class table*, and setting the bit to 1 for extant classes. For newly loaded classes (that are not in the closed-world) this bit is set to 0. During runtime we can query the class table to check if the object is extant or non-extant.
EST can also be performed on control flow predicates. Consider the following example:

```
boolean cond = bar();
if(cond)
{
  y = new_e T();  // create an extant object
  ...             // assume y is still points
                  // to extant object
}
else if(...)
  y = new_ne S(); // create a non-extant object
S20: y.foo();
```

In the above example, whenever `cond` is true, `y` points to an extant object. Therefore, `cond` can be considered as a surrogate for `EST(y)` at `S20`, and the part of program consisting of the "then" portion can be specialized with respect to `y.foo()` at `S20`.

5.1 Using the Sparse Evaluation Graph

We use the *sparse evaluation graph (SEG)* [10] for determining the extant tests and their placements. The SEG is a generalization of *static single assignment (SSA)* [13] and can be applied to both forward and backward monotone data flow problems, while offering the same benefits as those of SSA applied to forward def-use based data flow problems. Unlike SSA, where a definition triggers formation of ϕ-nodes, in the SEG only a statement with a non-identity transfer function for the analysis triggers formation of ϕ-nodes. For extant analysis, statements that affect the value of a receiver expression have a non-identity transfer function. The SEG has the following useful properties:

- Each use of a variable has a single definition point.
- ϕ-nodes are introduced to merge multiple definitions coming from distinct control flow paths. Let S be the set of definition points of a variable We introduce ϕ-nodes at the iterated dominance frontier $IDF(S)$.
- Given a variable v, let $SEG_d(v)$ denote the definition point of the use of v. $SEG_d(v)$ will dominate the use point of v.

Now, let χ be an exit boundary point, and v be the receiver expression at χ. The program point $m = SEG_d(v)$ will satisfy Properties 4.1 and 4.2 for the exit boundary point χ. Therefore, all program statements between m and χ inclusive can be safely specialized with respect to χ.
If the SEG definition point is the entry of a method, we can place the EST at each call site invoking the method instead of at the entry of the method. Placing the EST in a caller can enable further optimizations in the caller with respect to the called method.
Finally, since the runtime type of a compile-time object is determined at compile time in Java by the allocation site, we can regard compile-time objects with an identical runtime type as the same compile-time object. This will reduce the number of merge points and, thereby, allow the placement of EST earlier in the program.

5.2 Optimization of Extant Safety Tests

Let $\mathcal{PP}(m,\chi)$ be the set of program points which occur in one or more of the paths in $\mathcal{P}(m,\chi)$. Given two surrogates EST_m and EST_n of $EST(p_\chi)$, EST_m is *preferable* to EST_n if

$$EST_n \sqsubseteq EST_m \wedge \mathcal{PP}(n,\chi) \subseteq \mathcal{PP}(m,\chi).$$

For example, if $SEG_d(v)$ (described in Section 5.1) is a simple copy from a reference variable, such as "`v = w;`", the extant test applied to `w` at the program point $n = SEG_d(w)$ whose definition of `w` reaches the use of `w` at m is preferable to that applied to `v` at m. This process of identifying a preferable extant test can be repeated until the definition is a merge node (i.e., a ϕ node) or the definition is an assignment of a de-reference such as "`v = p.f1;`". Another instance of a favored EST is one that can cover multiple exit boundary points.
One can perform several optimizations to identify the most preferable among multiple ESTs. Partial Redundancy Elimination (PRE) for eliminating partially redundant ESTs is one example. An EST E_1 is partially redundant if the truth value of an earlier EST E_2 implies the truth value E_1 for certain program paths.
Using profiling information or static analysis, we can also hoist ESTs to infrequently executed program points. For example, a static analysis applied to the example code segment below might identify that all the objects pointed to by `p` at `S100` are extant if the object pointed to by `head` at `S1` is extant. In that case, we can place an extant test for the object pointed to by `head` at `S1` that covers all the objects pointed to by `p` at `S100` in the loop body.

```
. . .
S1:
for (T p = head; p != null; p = p.next)
{
    . . .
  S100:  p.m(...);
    . . .
}
```

There exists a tradeoff between the strength (i.e. precision) of an extant test and the size of the set of program points that can be specialized. Although $EST_n \sqsubseteq EST_m$, EST_n might be favored over EST_m if $\mathcal{PP}(m,\chi) \subseteq \mathcal{PP}(n,\chi)$ – the increased size of the specialized code, at the cost of the larger failure rate of EST_n, might still improve the overall performance of the optimized code. Profiling and static analysis can help determine whether (and where) the precision of an extant test can be sacrificed for the increased size of the specialized code.

6. IMPLEMENTATION AND EMPIRICAL RESULTS

Table 1 shows the benchmark programs used in our experiments. Except for **portBob** all benchmarks are from the SPECjvm98 suite. We performed two kinds of experiments: (1) measuring closed-world characteristics; and (2) measuring extant analysis characteristics. We used the Jalapeño Virtual Machine as our experimental platform. The Jalapeño Virtual Machine is an implementation of Java written in (mostly) pure Java [1].

Program	Description	aCWClass	CWMeth	aCWMeth
compress	Compression/Decompression	25	313	112
jess	NASA's CLIPS expert system	112	640	400
db	Database search and modify	16	328	104
javac	Source to byte code compiler	71	659	399
mpegaudio	Decompress audio file	55	479	270
mtrt	Multithreaded image rendering	38	460	235
jack	Parser generator generating itself	59	563	59
portBob	Portable Business Object Benchmark	46	751	398

Table 1: Descriptions of the Benchmarks Used in Our Experiments.

Program	Calls	CWCW	aCWCW	%CWCW	%aCWCW	VC	vCWCW	%vCWCW
compress	18162511	18162219	18156434	99.99	99.99	15752583	15752361	99.99
jess	5987201	5781028	5292904	96.55	88.40	5690075	5502619	96.71
db	2191950	2191541	81112	99.98	3.70	1552273	1551941	99.97
javac	3075125	2782215	1313793	90.47	42.72	2488027	2259527	90.81
mpegaudio	9434285	9417509	9245643	99.82	98.00	6227385	6227169	99.99
mtrt	23098433	22698823	20882723	98.26	90.41	21149934	20929658	98.95
jack	6177988	6177408	867514	99.99	14.04	4281738	4281478	99.99
portBob	3201292	3174413	1485197	99.16	46.39	1762623	1752009	99.39

Table 2: Closed-World Characteristics.

6.1 Closed-World Characteristics

We used a profile-based approach to construct the closed world. We first instrumented the Jalapeño Virtual Machine to collect method profile information (i.e., the number of times a method is executed) and construct the set of methods and classes included in the closed world. For the SPECjvm98 suite, we used the data size of 1 during the profiling and the closed-world construction phase. For **portBob** we used a data size needed for constructing a minimum population (please see [4] for details about **portBob**). We included methods that were executed at least once in the closed world. The closed world contains both the application methods and the Java library methods, but does not include methods from the Jalapeño Virtual Machine. The closed world for each of SPECjvm98 programs also includes the methods of the SPECjvm98 driver which executed at least once.

In Table 1 the aCWClass column indicates the number of classes in the application that contain at least one method in the closed world. The aCWMeth column indicates the number of methods in the application that were included in the closed world. The CWMeth column indicates the number of methods in the closed world, including Java library methods used (but not methods of Jalapeño).

To analyze the dynamic characteristics of the closed world, we again instrumented the Jalapeño Virtual Machine. For this we used the data size of 10 for SPECjvm98, and the population size of 10% for **portBob**. Table 2 shows the dynamic results that we collected. The column Calls shows the total number of method calls. The CWCW column shows the number of calls in which both the callee and the caller were in the closed world. The aCWCW column shows the number of calls in which both the caller and the callee are within the application.

For the SPECjvm98 suite on average, for 97.87% of the method calls, the caller and the callee are within the closed world (column %CWCW). For **portBob** this percentage is 99.16%. This suggests that one can perform optimistic interprocedural optimizations assuming the closed world is the whole program, and in large part the outside world does not pollute the closed world. The %aCWCW column in Table 2 shows the percentage of calls that are within the application. The percentage here varies from 3.7% to 99.99%. The benchmarks **db** and **jack** make a significant number of calls to Java library methods. For such programs, including Java libraries in the closed world gives better results.

The last three columns (VC, vCWCW, and %vCWCW) are numbers related to virtual calls. For both SPECjvm98 and **portBob** a large percentage of virtual calls are within the closed world.

6.2 Extant Analysis

We implemented extant analysis on the top of the Jalapeño Optimizing Compiler [1]. Tables 3, 4, and 5 show the results of our experiment. We used the closed-world program constructed during profiling for our analysis. We used method profile information to obtain dynamic counts. In all the tables the prefix "Dy" indicates a weighted dynamic count of the static information.

We implemented the extant analysis algorithm described in Section 3, assuming a root-connected closed world. We also make certain reasonable optimistic assumptions with respect to entry boundary points: (1) that public methods are not called from outside the closed world; and (2) the extant state of compile-time objects remains unchanged when they become reachable from static reference variables; and (3) when a reference variable is passed as a parameter to a method, the extant state of the compile-time objects reachable from the parameter remains unchanged. In the last case, however, we apply a meet with $\perp$ to the extant state of a compile-time object returned by the method. The closed-world characteristics described in the previous section, which show a high percentage of method calls to be in the closed world, support these optimistic assumptions with respect to entry boundary points.

Step 1, as stated in Section 3, is used to compute the extant

Program	VC	UNEVC	%UNEVC
compress	700	87	12.4
jess	1671	143	8.56
db	822	95	11.5
javac	2986	374	12.52
mpegaudio	988	86	8.70
mtrt	1645	92	5.59
jack	1735	105	6.05
portBob	2221	145	6.53

Table 3: Non-extant virtual call characteristics based on class hierarchy analysis.

state of the receiver expression for each invocation site. We implemented a simple class hierarchy analysis to construct the call graph. During the call graph construction, for a virtual call site $p.foo()$, it is checked whether the method $foo()$ is defined within the closed world in a class that is either the declared type P of p, or is an ancestor or a descendent class of P. If there is no such method $foo()$, then the receiver express p is marked as UCNE (unconditionally non-extant). Otherwise, the extant type of p remains UNKNOWN($\top$).

The column UNEVC in Table 3 shows the number of unconditionally non-extant virtual calls discovered by this analysis. On average about 8.98% of virtual calls are identified to be unconditionally non-extant.

We measured the number of virtual calls that are candidates for devirtualization based on their having one target method in the closed world. Let $p.foo()$ be a virtual call, and let D be the declared class of p. If there is exactly one implementation of $foo()$ in D or some ancestor or descendent class of D, then $p.foo()$ is a candidate for devirtualization. As a special case, if $foo()$ is implemented in class D and is declared as `final`, then the receiver expression is unconditionally extant and the call $p.foo()$ can be directly devirtualized. Table 4 illustrates these measurements. Column UnVC represents all virtual calls whose extant type is UNKNOWN($\top$) after the analysis of Step 1 described above. (UnVC is same as VC - UNEVC shown in Table 3). The column DeV indicates those calls in UnVC that are candidates for devirtualization. Column DyDeV represents the corresponding dynamic numbers. Column DeVF represents all calls in DeV that are unconditionally extant based on their being declared as `final`.

On average 91.20% of the UNKNOWN virtual calls are candidates for devirtualization and the corresponding dynamic percentage is 79.97%. Of the calls that are candidates for devirtualization, on average, 58.94% are unconditionally extant (the corresponding dynamic average is 61.40%). Thus on average 53.75% of the UNKNOWN virtual calls are unconditionally extant due to being declared as `final` (the corresponding dynamic number is 49.10%). These unconditionally extant calls can be directly devirtualized. For the remaining UNKNOWN virtual calls (i.e., 46.25% static percentage and 50.89% dynamic percentage), further analysis, described in the next section, is needed to determine their extant type.

6.3 Extant Analysis Using Pointer Analysis

We also implemented a simple flow-insensitive/context-insensitive pointer analysis to compute the extant type for all UNKNOWN receiver expressions, as described in Step 2 of Section 3. For pointer analysis we identified four kinds of relevant statements.

- $p = new_e T()$, an allocation site where extant objects are created;
- $p = new_{ne} T()$, an allocation site where non-extant objects are created;
- $p = q$, reference copy statement. We simplified $p.f = q$ and $p = q.f$ as $p = q$. This way we do not distinguish among objects accessed via object fields. We also do not distinguish among array components.
- Call statements. As stated in Section 6.2, we make an optimistic assumption that the extant state of compile-time objects reachable from a reference variable passed as a parameter remains unchanged in the callee.

6.4 Summary

Figure 4 summarizes how virtual calls in SPECjvm98 and **portBob** are classified. UNEVC in the figure is same as the UNEVC shown in Table 3. Here we analyze the pie chart for **portBob**. We can slice the 2221 virtual calls in **portBob** into six categories. As shown in Table 3, 6.5% of virtual calls are unconditionally non-extant. The label NDeV represents the number of virtual calls that have more than one target method within the closed world. Only 2.3% of the virtual calls have more than one target. The remaining 91.2% of the virtual calls are candidates for devirtualization. DeVF indicates the number of these calls that are unconditionally extant based on their being declared as `final`. There are 34.1% of such calls, which can be directly devirtualized. The remaining 57.1% of virtual calls are subject to pointer analysis for further classification. Pointer analysis determines that an additional 29.9% of all virtual calls are unconditionally extant (and so can be directly devirtualized), 0.1% are conditionally extant (devirtualization requires an EST), and an additional 27.1% are unconditionally non-extant.

Table 5 shows the extant characteristics for devirtualization based on this simple pointer analysis. Column DeVNF indicates those virtual calls that are candidates for devirtualization and are not unconditionally extant (as determined by the analysis of the previous section). For these virtual calls, pointer analysis is used to determine whether the receiver expression of such calls points to extant and/or non-extant objects. The columns UEDev, CEDeV, and UNEDeV, show which of these virtual calls are unconditionally extant, conditionally extant, and unconditionally non-extant, respectively.

Program	UnVC	DeV	%DeV	DyVC	DyDeV	%DyDeV	DeVF	%DeVF	DyDeVF	%DyDeVF
compress	613	566	92.33	15683774	15677145	99.96	389	68.73	15676404	99.99
jess	1528	1451	94.96	367189	291140	79.29	840	57.89	212151	72.87
db	727	662	91.06	58457	34020	58.19	417	62.99	26463	77.79
javac	2612	2362	90.43	193642	106550	55.02	1622	68.67	60921	57.17
mpegaudio	902	841	93.24	815313	635285	77.92	613	72.89	600129	94.47
mtrt	1553	1499	96.52	4705232	4345162	92.35	481	32.09	233123	5.36
jack	1630	1198	73.50	2154117	1755851	81.51	849	70.87	1139311	64.89
portBob	2076	2026	97.59	786832	751933	95.56	758	37.41	140235	18.65

Table 4: Devirtualization characteristics.

Program		DeVNF	UEDeV		CEDeV		UNEDeV	
			Raw	%	Raw	%	Raw	%
compress	St	177	43	24.29	6	3.39	128	72.32
	Dy	741	96	12.95	120	16.19	525	70.85
jess	St	611	268	43.86	88	14.40	255	41.73
	Dy	78989	34327	43.46	17223	21.80	27439	34.74
db	St	245	84	34.28	8	3.26	153	62.45
	Dy	7557	193	2.55	129	1.70	7235	95.70
javac	St	740	372	50.27	187	25.27	181	24.45
	Dy	45629	37264	81.66	3897	8.54	4468	9.79
mpegaudio	St	228	43	18.86	6	2.63	179	78.51
	Dy	35156	175	0.50	121	0.34	34860	99.16
mtrt	St	1018	812	79.76	6	0.59	200	19.65
	Dy	4112039	3974777	96.66	122	0.003	137140	3.33
jack	St	349	226	64.75	36	10.31	87	24.93
	Dy	616540	239825	38.89	274135	44.46	102580	16.64
portBob	St	1268	663	52.29	3	0.24	602	47.48
	Dy	746300	173721	23.28	39405	5.28	533174	71.44

Table 5: Devirtualization Extant Characteristics.

7. DISCUSSION AND RELATED WORK

In the Java community, there are two camps: the "virtual machine" camp that emphasizes the dynamic nature of Java such as dynamic class loading, and the "static compiler" camp that would like to apply whole program analysis and optimization for performance.[7] Supporting dynamic class loading for full Java compliance while applying static whole program analysis/optimization for performance has been viewed as an oxymoron, and adopting one has meant in large part abandoning the other. The extant analysis in the present work provides a means to accommodate this seemingly contradictory goal of full Java cmpliance and static whole program analysis/optimization. The more closely the $\mathcal{CW}$ matches the runtme characteristics of the application for an execution, the better the performance of the execution. The extant tests still ensure a correct execution, albeit with a poorer performance, if $\mathcal{CW}$ poorly matches the runtime characteristics of the execution.

We have used specialization of parts of the program as the principal mechanism for the optimizations based on extant analysis. Specialization is a technique for instantiating a program with respect to some runtime invariants [14; 12]. We apply extant analysis to determine the runtime invariants that we use in specializing methods and partial methods. One disadvantage of specialization is that it can increase code size and so should be applied only to "hot methods" [12].

[7]This observation is made by one of the PLDI reviewers.

Any static analysis and optimization, however, can be performed to the program parts that extant analysis finds unconditionally extant. Static analysis and optimization can also be applied to program parts found to be conditionally extant, by guarding the execution of the optimized code with dynamic extant tests. A tradeoff exists between the overhead due to dynamic extant tests and the improved performance by the optimization, which static analysis and/or runtime profiling can help analyze.

Extant tests offer better optimization opportunities than tests based on the runtime type of an object that typically guard against incorrect specialization for a receiver expression [20]. In contrast to such runtime tests, a single extant test can cover multiple statements, and thereby offers the opportunity for optimizations across the multiple statements. Further, the multiple statements covered by a single extant test can cross methods or class boundaries, in which case interprocedural optimizations such as inlining can be performed across multiple levels of method invocations.

Detlefs and Agesen [15] introduce the concept of *preexistence* in the context of a dynamic compiler so that inlining only takes place for those call sites for which it can be proved that the object pointed to by the receiver expression has been allocated. Their preexistence analysis is related to our intraprocedural extant analysis. One limitation of Detlefs and Agesen's scheme is that the inlining transformation in a method m may have to be invalidated and the method m may have to be recompiled for some future invocation of m. Their scheme could employ a simple dynamic extant

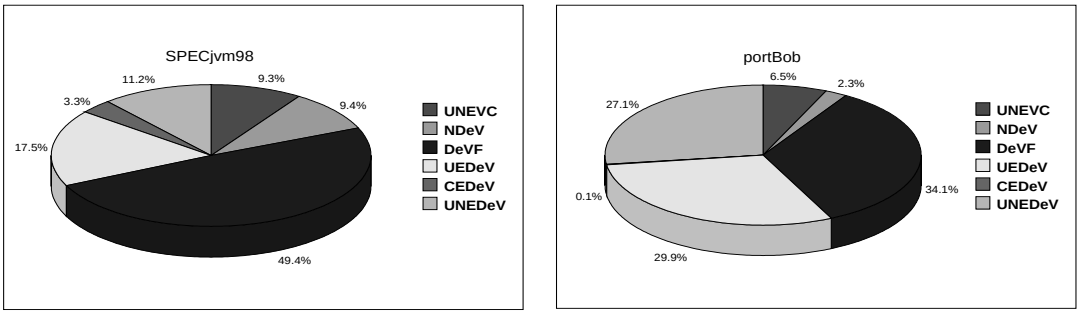

Figure 4: Virtual call characteristic for SPECjvm98 and **portBob**

test, based on the target of the object, to guard against incorrect execution of inlined code. Our framework is more general than preexistence and runtime type checking in that it can be applied to optimizations other than inlining, such as escape analysis.

Our formulation of parametric data flow analysis is related to other work. Burke and Torczon formulate a recompilation test that compares current interprocedural information with *annotation sets* that record those interprocedural facts which must continue to hold for a previous compilation to remain valid [5]. Their most precise computation of annotation sets involves augmenting data flow analysis to compute auxiliary information which is associated with the elements of a data flow solution. Chatterjee et al. parameterize points-to analysis for compiling large programs with multiple modules [8]. They obtain summary functions for points-to analysis of methods by inference of the relevant conditions on the unknown initial values for parameters and globals at method entry. Rountev et al. propose a framework for analyzing program fragments that is an extension of Chatterjee et al.'s work [29].

Static Java compilers which perform interprocedural analysis and optimization, such as HPCJ [22] and Marmot [17], do not allow dynamic class loading during program execution. JAX (Jikes Application eXtractor) [33] is a byte converter for compressing application class files. It performs whole program analysis, but again makes a "closed-world" assumption. Hotspot and other JIT compilers do not support aggressive interprocedural optimizations [27; 23]

8. CONCLUSIONS

In this paper we solve an important problem for efficient execution of Java: that of interprocedural optimization in the presence of dynamic class loading. We describe a framework for interprocedural optimization that does not depend upon runtime invalidation and recompilation. The framework is based on the optimization of a closed-world program prior to execution. A runtime safety test is used to enforce correctness. Our experimental results hold out the hope that with the framework described here, we can expect that a large percentage of a Java program can be optimized as if Java did not have the capability for dynamic class loading.

9. ACKNOWLEDGEMENTS

First and foremost we would like to thank Mark Wegman for his constant support and encouragement. We would like to thank Deepak Goyal, John Field, Harini Srinivasan, Ganesh Ramalingam, Vivek Sarkar, Rajesh Bordaw, and Peter Sweeney for their helpful comments on various drafts of this paper. We would like to thank Jalapeño team members, especially Derek Lieber, Dave Grove, and Steve Fink, for their help with the Jalapeño system. Finally, we thank the referees and the committee members of PLDI for their insightful comments.

10. REFERENCES

[1] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, D. Lieber, S. Smith, and T. Ngo. Implementing Jalapeño in Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.

[2] B. Alpern, M. Charney, J.-D. Choi, T. Cocchi, and D. Lieber. Dynamic linking on a shared-memory multiprocessor. In *International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[3] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, Oct. 1996.

[4] S. Baylor, M. Devarakonda, S. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. Munroe. Java server benchmarks. *IBM Systems Journal Special Issue on Java Performance*, 39(1), 2000.

[5] M. Burke and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[6] C. Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1992.

[7] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990. *SIGPLAN Notices 25(6)*.

[8] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, Jan. 1999.

[9] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, Jan. 1993.

[10] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *18th Annual ACM Symposium on the Principles of Programming Languages*, pages 55–66, Jan. 1991.

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

[12] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *In 1996 ACM Symposium on Principles of Programming Languages*, pages 145–156. ACM, January 1996.

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

[14] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 93–102, June 1995. *SIGPLAN Notices, 30(6)*.

[15] D. Detlefs and O. Agesen. Inlining of virtual methods. In *the 13 European Conference on Object-Oriented Programming*, pages 258–278, 1999.

[16] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994. *SIGPLAN Notices, 29(6)*.

[17] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for Java. Technical Report MSR-TR-99-33, Microsoft Research, June 1999.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[19] U. Holzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992. *SIGPLAN Notices* 27(6).

[20] U. Holzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 326–336, June 1994. *SIGPLAN Notices, 29(6)*.

[21] U. Holzle and D. Ungar. A third generation self implementation: Reconciling responsiveness with performance. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 229–243, 1994.

[22] IBM Corporation. IBM High Performance Compiler for Java, 1997. See `http://www.alphaWorks.ibm.com/formula`/hpc.

[23] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *ACM 1999 Java Grande Conference*, pages 119–128, June 1999.

[24] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992. *SIGPLAN Notices* 27(6).

[25] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, 1998.

[26] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1996.

[27] S. Meloan. The java hotspot[tm] performance engine: An in-depth look, 1999.

[28] NaturalBridge. BulletTrain optimizing compiler and runtime for JVM bytecodes, 1996. See `http://www.naturalbridge.com`.

[29] A. Rountev, B. Ryder, and W. Landi. Data flow analysis of program fragments. In *Proceedings of the 7th Symposium on the Foundations of Software Engineering*, 1999.

[30] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, Jan. 1998.

[31] V. Saraswat. Java is not type-safe, 1997. Information available in Web page at `http://www.research.att.com/~vj/bug.html`.

[32] B. Steensgaard. Points-to analysis in almost linear time. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 32–41, Jan. 1996.

[33] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Nov. 1999.

[34] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995. *SIGPLAN Notices, 30(6)*.