Automatic Program Specialization for Java

ULRIK P. SCHULTZ
Center for Pervasive Computing, University of Aarhus
JULIA L. LAWALL
DIKU, University of Copenhagen
CHARLES CONSEL
Compose Group, LaBRI/ENSEIRB

The object-oriented style of programming facilitates program adaptation and enhances program genericness, but at the expense of efficiency. We demonstrate experimentally that state-of-the-art Java compilers fail to compensate for the use of object-oriented abstractions in the implementation of generic programs, and that program specialization can eliminate a significant portion of these overheads. We present an automatic program specializer for Java, illustrate its use through detailed case studies, and demonstrate experimentally that it can significantly reduce program execution time. Although automatic program specialization could be seen as being subsumed by existing optimizing compiler technology, we show that specialization and compiler optimization are in fact complementary.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*optimization*; I.2.2 [**Automatic Programming**]: Program transformation

General Terms: Design, Experimentation

Additional Key Words and Phrases: Automatic program specialization, object-oriented languages, partial evaluation, Java, optimization

1. INTRODUCTION

Object-oriented languages encourage a style of programming that facilitates program adaptation. Encapsulation enhances code resilience to program modifications and increases the possibilities for direct code reuse. Method invocation allows program components to communicate without relying on a specific implementation. The use of these object-oriented abstractions in well-tested object-oriented designs (such as design patterns [Gamma et al. 1994]) leads naturally to the development of generic program components.

This program genericness is achieved, however, at the expense of efficiency. Encapsulation isolates individual program parts and increases the cost of data access. Method invocation is implemented using virtual dispatching; a virtual dispatch obscures program control flow and blocks traditional hardware and software optimizations. Compiler optimizations can eliminate some of these overheads [Agesen et al. 1993; Chambers and Ungar 1989; Dean et al. 1995; Detlefs and Agesen 1999; Grove et al. 1995; Hölzle and Ungar 1994; Ishizaki et al. 2000; Plevyak and Chien 1994;

Based on work done in the Compose Group at IRISA/INRIA, Rennes France; supported in part by Bull.
Authors' addresses: U.P. Schultz, DAIMI, Aabogade 34, DK-8200 Aarhus N, Denmark; J.L. Lawall, DIKU, Universitetsparken 1, DK-2100 Copenhagen, Denmark; C. Consel, LaBRI / ENSEIRB, 351 cours de la Libération, F-33405 Talence Cedex, France.

Sundaresan et al. 2000]. Nevertheless, invariants that can trigger optimizations may not be available to a compiler, or may not satisfy the compiler's optimization strategy, which must balance the cost of optimizations against the expected benefit across a wide variety of programs.

In many cases, the overheads due to genericness can be eliminated using *program specialization*. Program specialization is a software engineering technique that adapts a program to a given execution context. Information about this execution context can be provided by the programmer or derived from invariants present in the code. In contrast to compiler optimizations, program specialization is initiated explicitly by the programmer, and thus can adopt a very aggressive strategy for propagating such information. Specifically, within the object-oriented paradigm, program specialization simplifies the interactions that take place between program objects. The effect is to eliminate virtual calls, bypass encapsulation in a safe manner, and simplify invariant computations.

This paper presents an automatic program specializer for Java, named JSpec. JSpec combines interprocedural static analyses with aggressive global optimizations, which allows it to eliminate overheads due to the use of object-oriented abstractions in generic programs automatically. We describe the complete specialization process implemented by JSpec, including a novel use of aspect-oriented programming to represent specialized programs, and characterize the strengths and limitations of both automatic program specialization and the current implementation of JSpec. We then demonstrate experimentally that program specialization gives significant speedups when combined with state-of-the-art Java compiler technology.

Earlier work has (1) addressed the declaration of what to specialize, in the form of specialization classes [Volanschi et al. 1997], (2) described an early prototype of JSpec that generated specialized programs in C rather than Java [Schultz et al. 1999], (3) addressed the issue of selecting where to specialize, in the form of specialization patterns [Schultz et al. 2000], and (4) formalized specialization of programs in a small Java-like language [Schultz 2001]. As a next step in developing automatic program specialization for object-oriented languages, we present a complete Java-to-Java specialization process and our implementation, JSpec. Furthermore, we demonstrate experimentally the advantages of program specialization on a wide selection of generic programs. Lastly, we argue that automatic program specialization is not subsumed by existing optimizing compiler technology, but instead that the two are complementary.

Example. To illustrate how program specialization works, we use a small example written using the Strategy design pattern [Gamma et al. 1994]. Figure 1 shows a collection of four classes: `Binary`, `Add`, `Mult`, and `Power`. The abstract class `Binary` is the superclass of the two concrete binary operators `Add` and `Mult` (the strategies). The `Power` class can be used to apply a `Binary` operator a number of times to a base value, as illustrated by the following expression:

```
( new Power( y, new Mult() ) ).raise( x )
```

which computes x^y. The object diagram of this program is shown on the left side of Figure 2.

Consider the computation of x^3, illustrated on the middle of Figure 2. Invoking

```
abstract class Binary {
 abstract int eval(int x,int y);
 abstract int neutral();
}

class Add extends Binary {
 int eval(int x,int y) {
  return x+y;
 }
 int neutral() { return 0; }
}

class Mult extends Binary {
 int eval(int x,int y) {
  return x*y;
 }
 int neutral() { return 1; }
}
```

```
class Power {
 int exp; Binary op;

 Power(int exp,Binary op) {
  this.exp = exp;
  this.op = op;
 }

 int raise(int base) {
  int result = op.neutral();
  int e = exp;
  while( e-- > 0 )
   result = op.eval(result,base);
  return result;
 }
}
```

Fig. 1. Binary operators and a power function.

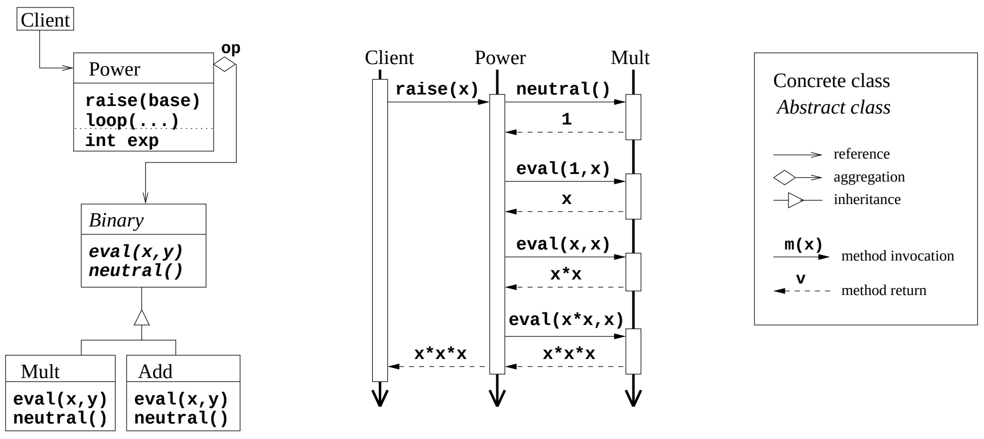


Fig. 2. Generic interaction with Power object.

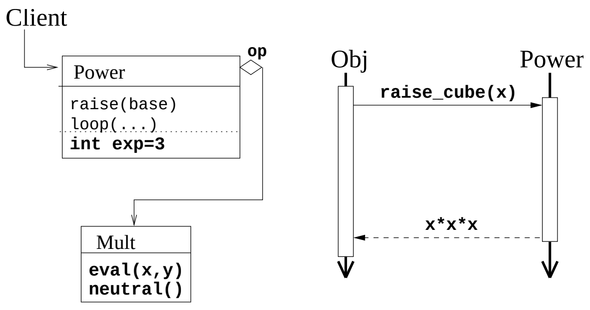


Fig. 3. Specialized interaction with Power object.

the method raise(x) of the Power object gives rise to a series of object interactions between the Power object and the Mult object that results in the return value x*x*x. To optimize for this case, we can manually enable the Power object to produce the result x*x*x directly. Specifically, we can add a method

```
int raise_cube( int base ) {
  return base * base * base;
}
```

to the class Power, and clients can use this new method to compute the result more efficiently. The object state on which this specialization is based and the resulting interaction are shown in Figure 3. Automatic program specialization can derive such a specialized method automatically, using constant propagation, loop unrolling, and virtual dispatch elimination.

The unspecialized program is also amenable to compiler optimizations. Because the Power object in the code above is only applied to a single kind of binary operator, standard compiler optimizations typically suffice to eliminate the virtual dispatch. When multiple operators are used with a given Power object, however, compilers generally do not remove the virtual dispatch to the eval method. Furthermore, an optimizing compiler does not usually propagate a constant value from an object field in order to perform loop unrolling, as done above. Indeed, as shown in the experiments reported in Section 6, specializing for the case where the Power object is used to compute a more complex expression (e.g., $5x^3$) produces a speedup of 4–11 times when compiled using state-of-the-art Java compilers. Specialization of other, larger, programs performs similar transformations and obtains comparable speedups (see Sections 5 and 6).

Overview. First, Section 2 gives an overview of automatic program specialization. Then, Section 3 describes how object-oriented programs can be specialized automatically, and Section 4 outlines the JSpec implementation. Section 5 presents two case studies showing how generic programs can be specialized using automatic program specialization. Section 6 describes a set of benchmark programs and presents the result of applying JSpec to these programs. Last, Section 7 discusses related work, and Section 8 presents our conclusion and discusses future work.

Terminology. The terminology used in the areas of object-oriented programming and program specialization overlap. In the area of object-oriented programming, class fields and class methods are sometimes referred to as static fields and static methods. The word “static” is however used in program specialization to indicate information known during the specialization phase. We thus always use the terms “class field” and “class method” rather than “static field” and “static method.” In addition, a subclass is often said to “specialize” its superclass. To avoid confusion, we refer to the relation between a subclass and its superclass in terms of inheritance (the subclass inherits from the superclass) or in terms of the subclass/superclass relation (one class is a subclass of some other class, or one class is the superclass of some other class).

2. BACKGROUND: AUTOMATIC PROGRAM SPECIALIZATION

Program specialization is a program transformation technique that optimizes a program fragment with respect to information about a context in which it is used, by generating an implementation dedicated to this usage context. One approach to automatic program specialization is *partial evaluation*, which performs aggressive inter-procedural constant propagation of values of all data types, and performs constant folding and control-flow simplifications based on this information [Jones et al. 1993]. Partial evaluation thus adapts a program to known (*static*) information about its execution context, as supplied by the user (the programmer). Only the program parts controlled by unknown (*dynamic*) data are reconstructed. Partial evaluation has been extensively investigated for functional languages [Bondorf 1990; Consel 1993], logic languages [Lloyd and Shepherdson 1991], and imperative languages [Andersen 1994; Baier et al. 1994; Consel et al. 1996].

In this paper, we only consider *off-line* partial evaluation [Jones et al. 1993]. This form of partial evaluation begins with an analysis known as *binding-time analysis*, which identifies the static and dynamic computations in a program based on information about which inputs can be considered static in the execution context. Based on the results of the binding-time analysis, the partial evaluator can specialize the program with respect to any concrete context that provides values for the static inputs. Specialization amounts to executing the static program parts on the known data, and reconstructing the dynamic constructs.

In contrast with most optimizing compilers, partial evaluation does not impose any bounds on the amount of computation that can be used to optimize a program. Specifically, specialization is not guaranteed to terminate, which frees the partial evaluator from making conservative choices to guarantee its termination. For example, a method invocation with all its arguments known is normally completely reduced during specialization. Nevertheless, partial evaluation relies on the user to direct the specialization process towards the program parts that contain worthwhile specialization opportunities, in order to avoid overspecialization (code explosion) and underspecialization (no benefit from specialization).

Because of the need for user control and because partial evaluation relies on complex analyses, in practice it often cannot be applied to complete programs. Instead, one typically extracts a specific slice from a program, specializes it, and then reinserts it into the original program [Consel et al. 1996]. An abstract description of the parts outside the program slice must be given to ensure correct treatment by the binding-time analysis. For an object-oriented program, the program parts to specialize and a description of the specialization context can be declared using the *specialization class framework* [Volanschi et al. 1997]. Furthermore, for a program written using design patterns, *specialization patterns* [Schultz et al. 2000] provide useful knowledge about how the program can be specialized. Section 5 illustrates the use of specialization classes and specialization patterns, and specialization patterns for the visitor and iterator design patterns are included in the appendix.

3. THE SPECIALIZATION OF OBJECT-ORIENTED PROGRAMS

We now describe our approach to automatic specialization of object-oriented programs. First, we explain how specialization transforms a program, then we address the issue of how to express a specialized program. Afterwards, we characterize those features of a program specializer that are essential for treating realistic programs, compare specialization to optimization, and last compare specialization to inheritance.

3.1 The specialization process

The execution of an object-oriented program can be seen as a sequence of interactions between the objects that constitute the program. Specifying particular parts of the program context can fix certain parts of this interaction. Program specialization simplifies the object interaction by evaluating the static interactions, leaving behind only the dynamic interactions. We refer to program specialization for object-oriented languages as *object-oriented-program specialization.*

Because objects interact by invoking methods, object-oriented program specialization is based on the specialization of method invocations. Such specialization optimizes the possible callees based on any static arguments, including the `this` argument. Conceptually, specialized methods are introduced into the receiver object under a new name. This transformation enables the object to respond to invocations of a new virtual method that can be used from other (specialized) methods.

Specializing a method optimizes the use of encapsulated values, virtual dispatches, and imperative computations:

Encapsulation. Data that are encapsulated inside an object can control computations elsewhere in the program. When such data are considered static by the specializer, they can be propagated to wherever they are used, and computations that depend on these data can be reduced. Specialization also eliminates the indirect memory references needed to access the data.

Virtual dispatching. The callee selection that takes place implicitly in a virtual dispatch can be seen as a decision over the type of the `this` argument. When the `this` argument is static, the decision of which method to invoke can be made during specialization. The callee method can then be specialized based on information about its calling context. Eliminating the virtual dispatch removes an indirect jump, which in turn simplifies the control flow of the program, and improves traditional compiler optimizations, as well as the branch prediction and pipelining performed by the processor. When the `this` argument is dynamic, each potential receiver method can be specialized speculatively on the assumption that it was chosen: each method is specialized to any static arguments but to a dynamic `this` argument. In this case, specialization generates a virtual dispatch to the specialized methods.

Imperative computations. Methods are specialized using transformations common to program specializers for imperative languages, such as constant propagation (of all data types), constant folding, conditional simplification, and loop unrolling. Specializing imperative computations is essential; in the Java specialization experiments reported in Section 6, most specialization scenarios require treating a mix of virtual dispatches, encapsulation, and imperative constructs.

In addition to the basic transformations outlined above, a program specializer can make further changes to the program, based on partially static data. For example, a partially static object (i.e., an object with a mix of static and dynamic fields) can in some cases be merged into its usage context. Such an object can be reduced to a set of local variables, similar to arity raising [Jones et al. 1993] and structure splitting [Andersen 1994], or reduced to fields fields of some enclosing object, similar to object inlining [Dolby and Chien 1998; 2000]. When the objects stored in an array have the same type, the array can be split so that each field is stored in a separate array [Budimlić and Kennedy 2001]. While transformations for optimizing the representation of partially static data can be essential for specializing some kinds of programs [Budimlić and Kennedy 1999; 2001; Veldhuizen 2000], they are not required for the experiments reported in this paper, and we consider the extension of our specialization process to include such transformations as future work.

3.2 The specialized program

To complete the specialization process, the specialized methods must be reintegrated into the existing class hierarchy. This reintegration must make each specialized method available at the call sites for which it was generated, and allow the specialized method to access any needed encapsulated data, without breaking encapsulation invariants.

One approach is to add each specialized method to its original class (i.e., the class defining its unspecialized counterpart) and to give the specialized method the original method's access modifiers. In this case, the receiver object also keeps its original class, which is sufficient to ensure that the receiver object contains the specialized method. This approach, however, has the undesirable effect of making the specialized method accessible to the rest of the program. Because specialization can eliminate safety checks, the use of a specialized method from unspecialized code can break encapsulation invariants. Furthermore, syntactically combining unspecialized and specialized methods in a single class definition obfuscates the appearance of the source program and complicates maintenance.

Specialized methods can be separated from existing methods by collecting the specialized methods derived from a single class in a new subclass, or by defining all of the specialized methods as methods of a single new class. The former approach implies that the receiver object of the specialized method call must be instantiated as an object of the new subclass; this transformation is not possible if the instantiation site is not part of the specialized program. The latter approach implies that the receiver object must be passed to the specialized method, essentially as an explicit `this` argument rather than an implicit one. This approach is also unusable when speculative specialization of a virtual dispatch is residualized as a virtual call, because virtual calls cannot be expressed using methods contained in a single class. Most importantly, neither of these approaches permits access to the private fields and methods of the original defining class.

A solution that separates the specialized method definitions syntactically from the source program but inserts these definitions into the right scope at compilation time is to express the specialized program as an aspect-oriented program. Aspect-oriented programming is an approach that allows logical units that cut across the program structure to be separated from other parts of the program and encapsulated into a separate module, known as an *aspect* [Kiczales et al. 1997]. We encapsulate the methods generated by a given specialization of an object-oriented program into an aspect, and weave the methods into the program during compilation. Access modifiers can be used to ensure that specialized methods can only be called from specialized methods encapsulated in the same aspect, and hence always are called from a safe context. Furthermore, the specialized code is cleanly

```
specclass Cube specializes Power {
 op: Mult; exp==3;
 @specialize: int raise(int base);
}
```

(a) Exponent and operator declared static

```
aspect Cube {
 introduction Power {
  int raise_cube(int base) {
   return base*base*base;
  }
 }
}
```

(b) Result of specializing for `Cube`

Fig. 4. Specialization of `Power` for exponent and operator static

separated from the generic code, and can removed from the program simply by deselecting the aspect.

3.3 The `Power` example revisited

Having outlined the specialization process, let us revisit the `Power` example of the introduction. In this example, the `Power` object is static, as are all of its fields. We can declare this specialization context using a specialization class. A specialization class provides specialization information about a given class, by indicating what fields are static and what methods to specialize. The specialization class `Cube` shown in Figure 4a declares that the fields of `Power` are static (the `this` argument is considered static by default), provides specific values for these fields, and indicates that the method `raise` should be specialized. Based on this information, specialization evaluates the invocation of the `neutral` method, unrolls the loop inside the method `raise`, and reduces the virtual dispatches to the method `eval`. The result of specialization is the aspect `Cube`, shown in Figure 4b (we use AspectJ syntax [Kiczales et al. 2001; XEROX 2000]). The aspect `Cube` declares the specialized method `raise_cube` using an `introduction` block; an `introduction` block specifies the name of a class and a list of methods to introduce into the class. Direct calls in `raise_cube` have been eliminated using method inlining, performed during the postprocessing phase of specialization.

We can also specialize the class `Power` for a context in which all fields are dynamic and the parameter `base` is static, as declared by the specialization class `PowerOfTwo` shown in Figure 5a. In this case, the `eval` methods are specialized speculatively with respect to the `base` value; the result after method inlining is the aspect `PowerOfTwo` shown in Figure 5b. (Figure 5b shows a simplified result; in Section 4 we describe the actual result of specialization in the presence of abstract methods.)

3.4 Essential program specializer features

In an offline partial evaluator, the set of constructs determined to be static by the analysis phase determines directly the benefit achieved by partial evaluation. In

```
specclass PowerOfTwo specializes Power {
 @specialize: int raise(int base),
  where base==2;
}
```

(a) Base value declared static

```
aspect PowerOfTwo {
 introduction Binary {
  abstract int eval_2(int x);
 }

 introduction Add {
  int eval_2(int x) { return x+2; }
 }

 introduction Mult {
  int eval_2(int x) { return x*2; }
 }

 introduction Power {
  int raise_pow2() {
   int result = op.neutral(), e = exp;
   while( e-- > 0 ) result = op.eval_2(result);
   return result;
  }
 }
}
```

(b) Result of specializing for `PowerOfTwo`

Fig. 5. Specialization of `Power` for base value static.

general, a more precise analysis can detect more static constructs. Nevertheless, extra precision also increases the complexity of the analysis, to the point that partial evaluation of some programs becomes infeasible. Thus, we must consider the ways in which values are typically used in object-oriented programs, to determine what degree of analysis precision is actually useful.

One aspect of the precision of an analysis is its granularity: what kinds of values are given individual binding times. Object-oriented programming is centered around the manipulation of objects, which are structured collections of values and methods. Because it is not possible for a static analysis to distinguish between all run-time objects, some approximation is needed. An option is to assign the same binding time to all instances of a given class, but because objects that are instances of the same class can play different roles in a program this very coarse-grained strategy does not provide enough precision. We have found that an adequate solution is to assign a single binding-time description to the set of objects created at each constructor call site, a feature that we name *class polyvariance.* An object itself typically contains multiple data values, and these values may play different roles within the program. Thus, rather than giving an object a single binding time, we use a compound binding time that contains a separate binding-time description of each of the fields. The granularity of the analysis also affects the analysis of methods and constructors. To maintain encapsulation invariants, object fields are often accessed using method invocations. Class polyvariance implies that a given method parameter can be bound to objects with different binding times. Thus, to realize the benefits of class polyvariance, each method invocation should also be analyzed individually, a feature that we name *method polyvariance*.

As an example of the need for this level of granularity, consider the definition of a vector and a dot-product operation shown in Figure 6a. The class `Vec` defines a vector of floating-point numbers. The class `VecOp` defines the method `dotP` that calculates the dot product of two objects of class `Vec`. Both classes are used in some context where the `dotP` method is passed two vectors, `x` and `y`. The vector `x` is initialized using static information, whereas the vector `y` is initialized using dynamic information. Only when these two instances of the same class are assigned separate binding times, can the static information in `x` can be exploited during specialization to yield the specialized program shown in Figure 6b. Moreover, since the data stored in the vectors is accessed using the `get` method, this method must be analyzed independently at each invocation.

Binding-time analyses for languages with imperative features typically uses an alias analysis to determine the locations that are read or written at each program point [Andersen 1994]. Since objects are heap-allocated and hence need to be tracked by an alias analysis, the granularity of the alias analysis limits the granularity of the binding-time analysis. To produce the specialized program shown in Figure 6b, the alias analysis must consider each instance of the class `Vec` to be an individual heap location and must analyze the `get` method independently for each of these locations.

Another aspect of the precision of an analysis is the number of descriptions assigned to each analyzed value. A single object can be used in both static and dynamic contexts within a program. Such a situation occurs, for example, when a static object is used both as an argument to an external method and within static expressions in the program slice to be specialized. This situation also commonly occurs when there are references to both static and dynamic fields of a static object. In both cases, the existence of a dynamic reference to the object implies that the object must be residualized in the specialized program, and thus considered dynamic. But giving the object the binding time dynamic makes it impossible to optimize the static uses. The solution is to give the object a compound binding time, *static and dynamic*, if the uses of the object warrant it, a feature known as *use sensitivity* [Hornof and Noyé 2000]. Individual references to the object are annotated static or dynamic according to their usage. Specialization creates one representation of the object for use during simplification of static constructs and another to residualize in the specialized program.

The benefit of giving an object both a static and dynamic binding time is illustrated by the slightly modified version of the dot product example shown in Figure 6c. Here, the object `x` is printed before it is passed to the dot-product operation. We do not want printing to take place during specialization, and hence the argument `x` must still be available in the residual program and thus have a dynamic binding time. By considering the object `x` to be static-and-dynamic, the

```
class Vec {                           class VecOp {
 float[] v;                            static float dotP(Vec x,Vec y) {
 Vec( float[] w ) { v=w; }              float res = 0.0;
 float get(int i) { return v[i]; }      for(int i=0; i<x.size(); i++)
 void set(int i,float d) { v[i]=d; }     res += x.get(i)*y.get(i);
 int size() { return v.length; }        return res;
}                                      }
                                      }
...
float[] fs = { 1.0, 0.0, 3.0 };
Vec x = new Vec(fs), y = ...;    // x static, y dynamic
float r = VecOp.dotP(x,y);
...
```

(a) Vectors, a dot product operation, and a usage context

```
aspect Vec_1_0_3 {
 introduction VecOp {
  static float dotP_1_0_3(Vec y) { return y.get_0() + 3.0*y.get_2(); }
 }
 introduction Vec {
  float get_0() { return v[0]; }
  float get_2() { return v[2]; }
 }
 ...
   Vec y = ...;
   float r = VecOp.dotP_1_0_3(y);
 ...
}
```

(b) Specialization with x static and y dynamic

```
Vec x = ..., y = ...;                     // x static, y dynamic
System.out.println(x+" and "+y);          // dynamic usage context
float r = VecOp.dotP(x,y);                // static usage context
```

(c) Use of static vector in dynamic and static context

```
Vec x = ..., y = ...;                     // x still initialized
System.out.println(x+" and "+y);          // x used
float r = VecOp.dotP_1_0_3(y);            // specialized; only y passed
```

(d) Specialization with x static-and-dynamic

Fig. 6. Specializing the vector dot-product operation

binding-time analysis can assign each use of `x` a static or dynamic binding time as needed, and the specialized program shown in Figure 6d can be produced. Here, the dot-product operation is specialized as before, even though `x` is needed in the residual program.

3.5 Specialization vs. compilation

Automatic program specialization and compilation rely on similar optimizations. However, compilers perform optimizations not considered in automatic program specialization, and automatic program specialization performs optimizations that are not performed by a typical compiler. Program specialization and compiler optimizations are thus complementary, and can even be synergistic.

Automatic program specialization uses type information to eliminate virtual dispatches; this technique is also often employed by aggressive compilers for object-oriented languages. *Customization* [Chambers and Ungar 1989], *selective argument specialization* (a generalization of customization) [Dean et al. 1995], and *concrete type inference* [Agesen et al. 1993; Dean et al. 1995a; Plevyak and Chien 1994; Wang and Smith 2001] all propagate type information about the `this` argument and the formal parameters of a method throughout the program, and can use this information to introduce new specialized methods into existing classes. The approaches differ, however, in how this type information is obtained when it is not actually explicit in the program. Customization and selective argument specialization can be used in conjunction with profiling to determine frequently occurring method argument types. Concrete type inference determines such information from static program analysis. In contrast, automatic program specialization relies on programmer-supplied invariants and aggressive static propagation of this information. Compiler techniques are useful when the programmer supplied information is insufficient, but automatic program specialization can exploit invariants that depend on an external usage context and are thus difficult to identify by automatic means.

Automatic program specialization also differs from compiler techniques in the degree to which it propagates inferred information. Program specialization propagates values of any type, including partially known objects, throughout the program and reduces any computation that is based solely on known information; no resource bound restricts the amount of simplification to be performed. Thus, a program specializer can achieve more pervasive optimization based on knowledge of a small set of configuration parameters than can a typical compiler. This degree of pervasiveness is justified by the fact that the programmer has determined that optimization with respect to this information is likely to be beneficial, and is tempered by the fact that the only optimization performed by automatic program specialization is the simplification of constructs that depend only on constants that can be determined during specialization. Compilers apply optimizations in a more restricted manner, with the goal of producing a program that performs well in a normal usage context, but apply a wider range of optimizations, such as copy propagation and loop invariant removal that do not necessarily depend on statically determined constants. Program specialization is thus dependent on a compiler for traditional intra-procedural optimizations such as copy propagation, common subexpression elimination, loop invariant removal, etc. that are essential for good performance. Furthermore, optimizations that are not expressible at the language level, such as register allocation and array bounds check elimination, cannot be performed by a program specializer, and must be handled by a compiler.

The characteristics of selective argument specialization, concrete type inference,

●=yes ○=no ◒=partially	Selective argument specialization	Concrete type inference	Automatic program specialization
Optimizes megamorphic call points	◒	○	●
Generally applicable	●	●	○
Specializes imperative parts	○	○	●
User control	○	○	●
Bounded resources	●	●	○
Applicable at run time	●	●†	●*
Whole-program assumption	○	●	◒
Verifiable result	○	○	●

Table I. Comparison of techniques. (†): For example, when using a technique such as CHA. (*): Not supported currently by JSpec — see Section 8.

and automatic program specialization are summarized in Table I. The first part of the table analyzes the overall applicability of each technique: what can be optimized, where is the optimization technique applicable, and whether the technique optimizes imperative as well as object-oriented constructs. As we have described, program specialization is less generally applicable than compiler optimization techniques, but is also more aggressive. The second part of the table assesses the degree of user interaction required. Partial evaluation is particularly distinguished from the other techniques in that the user can both control the information on which optimization is based, and inspect the result of the optimization (in the form of binding-time annotations and specialized source code) to verify whether this information was exploited in a useful manner.

To further improve object-oriented-program specialization, it is likely that ideas from selective argument specialization and concrete type inference can be integrated into the specialization process. For example, automatic program specialization can perform concrete type inference during specialization [Braux and Noyé 2000], which could be used to reduce virtual dispatches with a dynamic `this`. We leave such improvements as future work.

3.6 Specialization vs. inheritance

As mentioned in the introduction, program specialization is different from the object-oriented notion of inheritance. Where inheritance (normally) adds new state and new behavior, specialization of a program makes state constant and simplifies behavior. Furthermore, inheritance refines one or more classes into a single new class, which is ill-suited to expressing the specialization of an entire program. Last, inheritance describes compile-time relations between the classes of the program, whereas program specialization is based on execution of parts of the program.

4. THE JSPEC SPECIALIZER

JSpec is an automatic program specializer for Java that has been implemented according to the principles presented in the previous section. We now give an overview of JSpec, describe how JSpec specializes features specific to Java, and last characterize the limitations of JSpec.

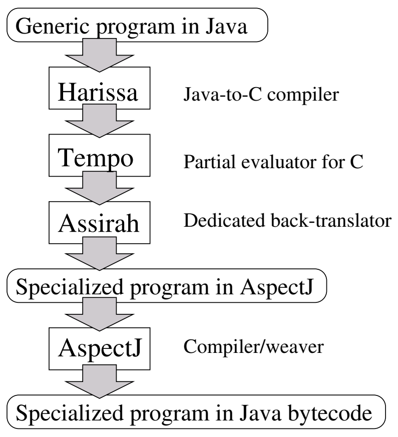


Fig. 7. JSpec specialization process

4.1 Overview of JSpec

JSpec is an off-line automatic program specializer for the Java language that integrates a wide range of state-of-the-art analyses and specialization features, and offers several input and output language options:

—JSpec treats the entire Java language, excluding exception handlers, reflection and `finally` regions. It takes as input Java source code, Java bytecode, and native functions.

—The JSpec binding-time analysis is context-sensitive, class-polyvariant (each object creation site is assigned a binding time individually), use-sensitive [Hornof et al. 1997], and flow-sensitive.

—Specialized programs can be generated either as Java source code encapsulated in an AspectJ aspect [Kiczales et al. 2001; XEROX 2000], as C source code for execution in the Harissa environment [Muller and Schultz 1999], or as binary code generated at run time for direct execution in the Harissa environment.

—JSpec can be applied to a user-selected program slice, which allows time-consuming analyses and aggressive transformations to be directed towards critical parts of the program.

In this paper we only consider Java-to-Java specialization; we refer to earlier work [Schultz et al. 1999] and the first author's PhD dissertation [Schultz 2000] for more information on using C as the input or output language.

JSpec has been designed with an emphasis on re-use of existing technology. In particular, JSpec uses C as an intermediate language and uses a specializer for C programs, named Tempo [Consel et al. 1996], as its partial evaluation engine. This approach allows us to take advantage of the advanced features found in a mature partial evaluator.

Figure 7 illustrates the complete specialization process. To specialize a Java program, the Java-to-C compiler Harissa [Muller and Schultz 1999] is first used to translate the program into C. The program is then specialized using Tempo, and a dedicated back-translator named Assirah maps the C representation of the specialized Java program back into Java. Finally, AspectJ is used to weave the specialized Java program with the original program to produce a complete, specialized program.

C is used as an intermediate language as follows. Java objects are represented as C structures with a compatible layout between a class and its superclass. Thus, an object field access becomes a C structure field access. A virtual dispatch becomes an indirect C function call through a virtual table accessible as a special object field. After specialization, auxiliary information that describes how Java fields and methods are named in the C program is used to map the residual C code back into Java code.

4.2 Specialization of object-oriented features

Specialization consists of propagating static values across method and constructor calls, performing simplifications according to these values, and constructing a specialized program consisting of the constructs that depend on dynamic information. The processes of propagating static values and performing simplifications can be carried out for Java programs in a manner similar to that used for specializing imperative programs. Constructing the specialized program is complicated by the fact that Java has a richer set of “procedures” than imperative languages, including methods declared in concrete classes, methods declared in abstract classes or interfaces, and constructors. In treating these cases, some care must be taken to respect the constraints of the Java language.

We first consider the treatment of an instance method declared in a concrete class. In the intermediate C representation used by JSpec, a Java instance method is represented as a procedure having the receiver object as its first argument. The strategy for translating a specialized variant of such a method back to Java is determined both by the specialization of the method definition and by the specialization of the receiver object. There are several cases, depending on how much is known about the receiver object at specialization time.

If the receiver object is known at specialization time and the specialized method is independent of any dynamic data contained in the receiver object, then specialization eliminates the receiver object argument. Because the specialized method does not refer to any instance variables of the class in which its original definition appears, its definition is added, using an aspect, to this class as a class method, and the specialized call is implemented as a call to this class method.

If the receiver object is known at specialization time, but contains some dynamic data that is referenced by the specialized method definition, then the specialized method definition is added, using an aspect, to the class of the source method as an instance method. This approach provides the specialized method definition with access to the data in each runtime instance of the receiver object. For efficiency, we would also like to inform the compiler that the specialized definition of the method is the only one that will exist at run time, thus allowing the compiler to implement the call as a direct call rather than as a virtual call. For this purpose, the specialized method is declared as a `final` method, which cannot be overridden, and a downward cast is used at the call site to coerce the receiver object to the class declaring the specialized method. This strategy replaces the cost of a virtual call by the cost of a type case. A similar strategy has been taken in the context of object-oriented compilers that perform optimizations based on type feedback or Class Hierarchy Analysis (CHA) [Aigner and Holzle 1996; Dean et al. 1995b; Hölzle and Ungar 1994]. Because the Java-to-C translator, Harissa, used by JSpec contains a CHA, this strategy is also applied whenever Harissa determines that only one method definition can be referenced by a virtual call, even when the receiver object is dynamic.

If the receiver object is dynamic and there is more than one possible callee, then JSpec creates a specialized variant of each possible method definition. Each specialized method is added, using an aspect, to the class of its corresponding original definition, and the virtual call is reconstructed in the specialized program. In this case, if the source program declares the original method as an abstract method of an abstract class or in an interface, the specialized method must be additionally declared in the same manner. If the specialized method is to be declared in an abstract class, several solutions are possible. In the example of Section 3 (Figure 5), we declare the specialized method `eval_2` as an abstract method. This solution, however, requires that all concrete subclasses define the specialized method, including those that are not part of the program slice to be specialized. A less restrictive solution, which JSpec uses, is to define a concrete method that always generates an error in the abstract class. Specialized variants of the method then override this definition. If the specialized method is to be declared in an interface, the solution of using a concrete method is not applicable. Because inheritance from multiple interfaces is possible, however, we simply define a new interface declaring the specialized method, and extend the possible classes of the receiver object to implement the new interface. The receiver object is then cast to the new interface type at the call to the specialized method.

Figures 8 and 9 use a small hierarchy of binary operators to illustrate the treatment of abstract classes and interfaces. In both cases, the method `f`, which computes the result of applying a binary operator to two integer inputs, is specialized with respect to a dynamic `Op` object as the receiver object and a static value (0) of the argument `y`. In Figure 8b, the specialized program introduces a new specialized method `f_0` into the `Op` abstract class and its subclasses; the method in the `Op` class always gives an error, whereas the methods that override it use the specialized definitions. In Figure 9b, the specialized program introduces a new interface `IxI2I_0` and and uses introduction blocks to add the interface to the appropriate class declarations.

Following the same strategy as is used for a specialized method, a specialized constructor is added, using an aspect, to the class defining the original constructor. Because a class can only define multiple constructors through overloading, JSpec creates a new, empty class for each specialized constructor, and adds a dummy argument of this type to the constructor, to distinguish it from the other constructors of the class. Specialization must also preserve the chain of superclass constructor calls. The constructors in such a chain are either all accessible to the specializer, and thus all specializable, or the chain terminates in a constructor that is external to the targeted program slice. Specialization generates a parallel chain of constructors, terminating with an invocation of an external constructor, if any.

```
abstract class Op { abstract int f(int i,int j); }
class Add extends Op { int f(int i,int j) { return i+j; } }
class Mul extends Op { int f(int i,int j) { return i*j; } }
class Use { int apply(Op p,int x,int y) { return p.f(x,y); }
```

(a) Generic program

```
aspect ZeroArgument {
 introduction Op { int f_0(int i) { throw new JSpecExn(); } }
 introduction Add { int f_0(int i) { return i+0; } }
 introduction Mul { int f_0(int i) { return i*0; } }
 introduction Use { int apply_0(Op p,int x) { return p.f_0(x); }
}
```

(b) Specialized program

Fig. 8. Specialization in the presence of abstract methods

```
interface IxI2I { int f(int x,int y); }
class Add implements IxI2I { int f(int x,int y) { return x+y; } }
class Mul implements IxI2I { int f(int x,int y) { return x*y; } }
class Use { int apply(IxI2I p,int x,int y) { return p.f(x,y); } }
```

(a) Generic program

```
aspect ZeroArgument {
 interface IxI2I_0 extends IxI2I { int f_0(int x); }

 introduction Add {
  implements IxI2I_0; int f_0(int x) { return x+0; }
 }

 introduction Mul {
  implements IxI2I_0; int f_0(int x) { return x*0; }
 }

 introduction Use {
  int apply_0(IxI2I p,int x) { return ((IxI2I_0)p).f_0(x); }
 }
}
```

(b) Specialized program

Fig. 9. Specialization of interface calls

4.3 Limitations of JSpec

JSpec is a partial evaluator, and is thus limited by the inherent restrictions of partial evaluation. For partial evaluation to optimize a program, it must be possible to separate the program execution context into static and dynamic parts such that the overheads caused by genericness can be considered static. Furthermore, throughout the parts of the program that are to be specialized, dynamic and static values must remain clearly separated, so that the static values do not become dynamic. In practice, programs often must be modified to ensure this separation [Jones et al. 1993]. Alternatively, programs can be written with specialization in mind; specialization patterns can be followed to ensure that the finished program specializes well.

Java offers non-object features, such as exceptions, multi-threading, dynamic loading, and reflection. These features are treated in a minimal way by JSpec. JSpec does not support exception handlers in the program slice being specialized, and therefore considers a `throw` statement to terminate the program. JSpec only specializes a single thread of control, and speculatively evaluates code inside synchronized regions; while not appropriate in all situations, this approach offers a simple and predictable behavior. Dynamic loading is essentially orthogonal to JSpec. JSpec requires that all of the classes that are referenced by the code slice be included in the code slice itself or be considered external code and be described abstractly. How such classes become available at run time is irrelevant. JSpec does not support reflection in the program slice being specialized, and relies on the programmer to describe the side effects that can be performed using reflection in the external code.

The alias analysis in JSpec is monovariant, which implies that the binding-time analysis is performed using alias information that has been merged for all invocations of each method. The use of a monovariant alias analysis can thus reduce the precision of the binding-time analysis and impede the specialization of object-oriented programs, as was explained in Section 3.4. Nevertheless, a polyvariant alias analysis would often generate large numbers of alias variants of methods that have identical binding-time properties and thus specialize identically. Ideally, alias variants should be generated on-the-fly to match different binding times, similarly to procedure cloning [Cooper et al. 1992]. As a simple solution, JSpec relies on user-supplied information to identify classes that contain methods for which a high degree of precision is needed, and clones such methods for each call site.

The current implementation of JSpec does not automatically generate the code needed for transparent reintroduction of specialized code into a program. Specialization classes allow transparent reintroduction of code, but only in simple cases [Volanschi et al. 1997]; an extension of specialization classes to treat more complex cases, as used in this paper, is currently under development. For this reason, the call to the specialized entry point needs to be created manually.

5. CASE STUDIES

We now present two case studies that illustrate how object-oriented programs can be specialized using automatic program specialization. We first describe the specialization of a program written using the visitor pattern [Gamma et al. 1994], and then the specialization of a part of the OoLaLa object-oriented linear algebra library [Luján et al. 2000]. In both cases we use specialization patterns and specialization classes to identify and describe specialization opportunities, and in both cases automatic program specialization leads to significant speedups (experimental results are reported in the next section).

In addition to the case studies presented in this section, previously published work describes the specialization of programs written using an image processing framework [Schultz et al. 1999], a set of programs written using the builder, bridge and strategy design patterns [Schultz 2000; Schultz et al. 2000], programs written using a GUI framework in the style of the Java JDK 1.1 AWT [Schultz 2000], and programs written using a checkpointing library [Lawall and Muller 2000].

5.1 The visitor design pattern

The visitor design pattern is a way of specifying an operation to be performed on the elements of an object structure externally to the classes that define this structure [Gamma et al. 1994]. In a language without multi-dispatching, such as Java, the visitor design pattern is implemented using a technique referred to as *double-dispatching*, where the first dispatch selects the kind of object structure element to operate on, and the second dispatch selects the visitor to use. Given a specific visitor, the second virtual dispatch becomes fixed, and thus can be removed by specialization.

As an example of a use of the visitor design pattern, we implement a simple binary tree structure, and use the visitor pattern to implement operations on the tree structure. Figure 10a shows the class diagram for this example. The class `Tree` is the abstract superclass of the concrete classes `Leaf` (which holds an integer value) and `Node` (which has two children). The class `TreeVisitor` is the abstract superclass of the concrete visitors that operate on the tree; in the example, we use the subclass `FoldBinOp`, which folds an operator of class `Binary` (from Section 1) over the tree. Excerpts of the implementation are shown in Figure 10b. Figure 10c illustrates parts of the interaction between a `FoldBinOp` visitor and a tree that consists of a node and two leaves.

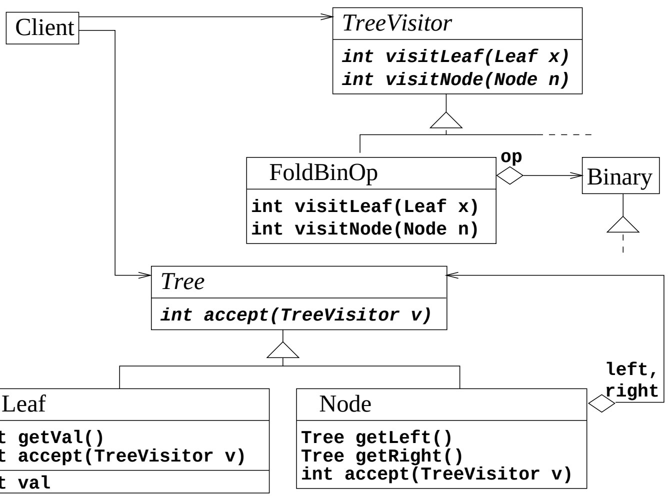
The visitor specialization pattern (see Appendix) suggests that when the choice of the visitor is fixed, automatic program specialization can eliminate the double dispatching that takes place between the visitor and the objects that it traverses. We use a specialization class to declare the specialization context of the program:

```
specclass FoldPlus specializes Client {
 @specialize: int processTree(Tree t,TreeVisitor v),
  where v: FoldWithPlus;
}

specclass FoldWithPlus specializes FoldBinOp {
 op: Plus;
}
```

The specialization class `FoldPlus` declares that specialization should begin at the `processTree` method of the class `Client`. The second argument of `processTree` is declared to be specialized according to the `FoldWithPlus` specialization class, which indicates that `v` is a `FoldBinOp` object in which the `op` field, representing the operator to apply, is a `Plus` object. Specialization simplifies the virtual dispatches inside the `accept` methods of the `Node` and `Leaf` classes into direct calls, speculatively specializes any calls to `accept` inside the `visit` methods (with `Node` and `Leaf` as potential receivers), and reduces the virtual dispatch to the binary operator. Figure 11a shows a fragment of the specialized program after method inlining. Specialization simplifies the interactions shown in Figure 10b into the interactions shown in Figure 11b.

In the simple case where the program uses only a single visitor, a compiler can usually perform similar optimizations. However, when multiple visitors are used, the virtual dispatches to the `visitNode` and `visitLeaf` methods inside the `Node` and



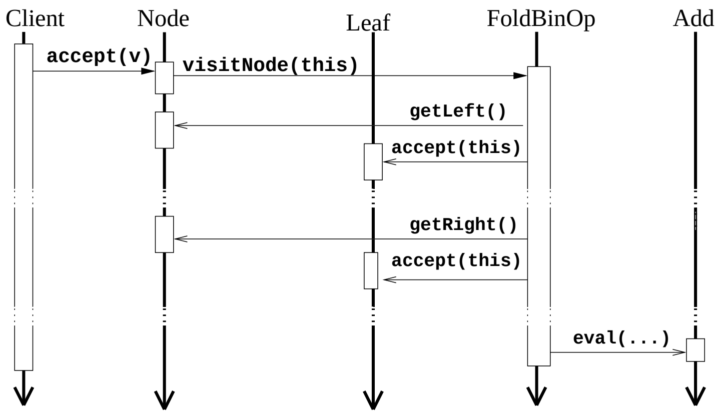
(a) Class diagram

```
class Client {
 int processTree(Tree t,
                TreeVisitor v) {
  return t.accept(v);
 }
}

class Node extends Tree {
  ...
 int accept( TreeVisitor v ) {
  return v.visitNode(this);
 }
}
```

```
class FoldBinOp extends TreeVisitor {
 ...
 int visitNode( Node n ) {
  return this.op.eval(
   n.getLeft().accept(this),
   n.getRight().accept(this) );
 }
}
```

(b) Extracts of the source code



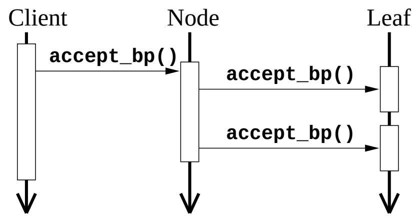
(c) Interaction diagram

Fig. 10. Visitor, generic program

```
aspect FoldPlus {
 introduction Client {
  ... t.accept_bp() ...
 }
 ...

 introduction Node {
  int accept_bp() {
   return this.left.accept_bp() + this.right.accept_bp();
  }
 }
}
```

(a) Specialized program



(b) Specialized interaction

Fig. 11. Visitor, specialized program

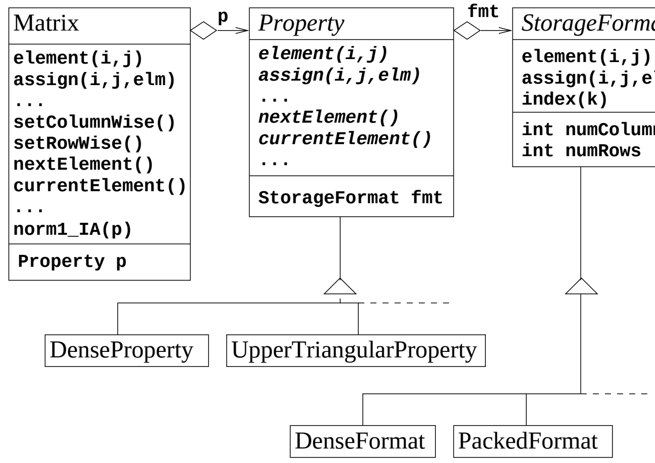
`Leaf` classes have multiple receivers, and cannot easily be removed.

5.2 The OoLaLa linear algebra library

The OoLaLa linear algebra library has been designed according to an object-oriented analysis of numerical linear algebra [Luján et al. 2000]. Compared to traditional linear algebra libraries, OoLaLa is a highly generic, yet simple and streamlined, implementation. However, as the designers point out, the genericness comes at a cost in terms of performance. In this section, we use automatic program specialization to map matrix operations implemented at a high level of abstraction to an efficient implementation.[1]

In the OoLaLa library, matrices are classified by their mathematical properties, for example dense or sparse upper-triangular. A matrix is represented using three objects from different class hierarchies, as illustrated in Figure 12a. The class `Matrix` acts as an interface for manipulating matrices, by delegating all behavior specific to mathematical properties to an aggregate object of class `Property`. Subclasses of the abstract class `Property` define, for example, how iterators traverse matrix elements (e.g., by skipping zero elements in sparse matrices). The `Property` classes delegate the representation of the matrix contents to an object of class `StorageFormat`. The concrete subclasses of the abstract class `StorageFormat` all store the matrix elements

[1]Because the source code of the OoLaLa library is not available, we have reimplemented parts of this library for our experiments. The implementation follows the description of OoLaLa found in Luján, Freeman and Gurd's paper [Luján et al. 2000] and in Luján's Master's Thesis [Luján 1999].



(a) Structure of generic program

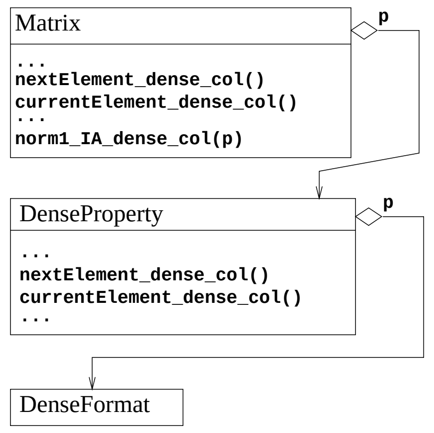
```
class Matrix {
 ...
 static double norm1_IA(Property p) {
  double sum;
  double max = 0.0;

  p.setColumnWise();
  p.begin();
  while( !p.isMatrixFinished() ) {
   sum=0.0;
   p.nextVector();
   while( !p.isVectorFinished() ) {
    p.nextElement();
    sum+=Math.abs(p.currentElement());
   }
   if(sum>max) max=sum;
  }
  return max;
 }
}

class DenseProperty extends Property {
 ...
 void nextElement() {
   this.current+=this.stride;
   this.minorCounter--;
 }
}
```

(b) Source code for generic program

Fig. 12. Generic version of a fragment of the OoLaLa library



(a) **Structure of specialized program**

```
aspect DenseNorm {
 ...
 introduction Matrix {
  ...
  static double norm1_IA_dense(Property p) {
   double sum;
   double max = 0.0;

   DenseProperty d=(DenseProperty)p;
   d.setColumnWise_dense();
   d.begin_dense_col();
   while( !d.isMatrixFinished_dense_col() ) {
    sum=0.0;
    d.nextVector_dense_col();
    while( !d.isVectorFinished_dense_col() ) {
     d.nextElement_dense_col();
     sum+=Math.abs(d.currentElement_dense_col());
    }
    if(sum>max) max=sum;
   }
   return max;
  }
 }
 ...

 introduction DenseProperty {
  final void nextElement_dense_col() {
   this.current+=1;
   this.minorCounter--;
  }
  ...
 }
}
```

(b) **Source code for specialized program**

Fig. 13. Specialized version of a fragment of the OoLaLa library

in a one-dimensional array, and define a mapping from ordinary matrix coordinates to an index in this array. This decoupling of a single matrix into three objects from separate class hierarchies is a use of the bridge design pattern [Gamma et al. 1994].

Figure 12b shows the implementation of the matrix operation norm1, written $\| \cdot \|_1$. For a matrix A, the matrix operation $\|A\|_1$ computes the maximum of the norms of the column vectors of the matrix A. This operation is implemented easily using matrix iterators, as shown by the method `norm1_IA` of class `Matrix`. The method first calls `a.setColumnWise()` to select column-based iteration. The outer loop then extracts each column vector, which is traversed by the inner loop.

Figure 12b also shows part of the definition of the class `DenseProperty`, specifically the method `nextElement`, which is used to implement iteration over the elements of a dense matrix. An iterator can traverse the matrix either by row or by column. In the case of dense matrices, column-based traversal means that the underlying one-dimensional array is traversed with increments of 1, and row-based traversal means that the underlying array is traversed with increments equal to the height of the matrix. The increment used in the traversal is referred to as the *stride* [Blount and Chatterjee 1999], and is stored in a field of `DenseProperty`.

The bridge specialization pattern [Schultz 2000] suggests that when the representation is known, automatic program specialization can eliminate virtual dispatches between the interface object and the representation object. Furthermore, the iterator specialization pattern (see Appendix) suggests that when the structure that the iterator traverses is known and the program manipulates the iterator in a fixed way, then automatic program specialization can replace uses of the iterator method by direct manipulation of the structure. Again, we use specialization classes to declare the specialization context of the method `norm1_IA`:

```
specclass DenseNorm specializes Matrix {
 @specialize: void norm1_IA(Property a),
  where a: DenseWithDense;
}

specclass DenseWithDense specializes DenseProperty {
 fmt: DenseFormat;
}
```

These classes declare that the `Property` argument to the norm operation is a dense matrix that uses the dense representation format.

Specialization simplifies the virtual dispatches inside the `norm1_IA` method into direct calls, simplifies the virtual dispatches inside the methods of class `DenseProperty` into direct calls, and optimizes the iterator to perform only column-based traversal. Figure 13a illustrates the structure of the specialized program. A specialized `norm1` operation and specialized iterator methods have been added to the classes of the program. As an example of how the iterator methods are optimized for column-based traversal, the specialized `nextElement` method no longer accesses the field `stride`, but directly increments the field `current`. Figure 13b shows the specialized source code; to enhance readability, the `Property` object is cast once to the type `DenseProperty`, rather than at each use as is actually the case. JSpec does not redeclare the fields `p` and `fmt` to have the more specific types indicated in the specialization classes `DenseNorm` and `DenseWithDense`, respectively. Thus, type casts are inserted in the specialized code whenever the contents of these fields are accessed.

As was the case for the visitor pattern example, in the simple case where the program uses only a single kind of matrix representation, a compiler can usually perform similar optimizations. However, when the program uses multiple matrix representations, virtual dispatches from `Matrix` objects to `Property` objects and from `Property` objects to `StorageFormat` objects cannot easily be removed. In addition, optimizing the iterator methods for column-based traversal goes beyond the optimizations normally performed by a compiler.

6. EXPERIMENTAL STUDY

In this section, we compare the execution time of generic programs to the execution time of specialized programs. With the exception of the benchmark programs classified as "imperative" below, the programs are written using object-oriented abstractions wherever appropriate. The programs are compiled using a selection of state-of-the-art Java compilers. Our goal is to show that optimizing compilers at most only partially eliminate the genericness of these benchmark programs; propagation of specific invariants, as performed by program specialization, is needed to more completely optimize a generic program for a given usage context.

6.1 Benchmark programs

To assess the performance improvements due to partial evaluation, we consider a wide selection of benchmark programs written in a generic style, as summarized in Table II. We do not use standard benchmark, because programs from standard benchmark suites (e.g., SpecJVM98 [SPEC 1998]) usually either contain no opportunities for specialization or are structured in a way that is incompatible with specialization. The programs are grouped by the primary kind of specialization opportunity they expose, namely *imperative*, *object-oriented*, or *mixed*. All benchmark programs are computationally intensive. They do not perform a large amount of I/O, do not allocate a large amount of memory, and do not contain multi-threaded code. The size of the program slice targeted with JSpec ranges from roughly 25 lines (Power) to roughly 1100 lines (ChkPt) of Java source code.

Imperative opportunities. Some object-oriented programs are primarily imperative in nature, although they may benefit from object-oriented constructs to provide structuring or data encapsulation. In each program, part of the primitive-type data in the execution context is static.

FFT: Computes a one-dimensional Fast-Fourier Transform on data stored in an array; the benchmark is taken from the Java Grande benchmark suite [Java Grande Forum 1999]. The algorithm is parameterized by the radix size. For specialization, the radix size is static, and the transformed data is dynamic. We specialize for three different radix sizes, 16, 32, and 64. Specialization unrolls all loops and eliminates all trigonometric computations.

Romberg: Romberg integration approximates the integral of a function on an interval using estimations. The algorithm is parameterized by the number of iterations, the interval characteristics, and the function. For specialization, the number of iterations used in the approximation is static, and all other data is dynamic. Specialization unrolls loops and eliminates numerical computations.

	Name	Description	Slice size bytecode	Slice size lines	Genericness	Static input
(imperative)	FFT	Fast-Fourier Transform	915 B	148	varying radix size	radix size: 16, 32, 64
	Romberg	Romberg integration	774 B	50	varying number of iterations	number of iterations: 2
(object)	Builder	dense and sparse matrices	1463 B	282	representation-independent multiplication algorithm	representation
	Iterator	set membership and intersection	717 B	205	varying underlying data-structure implementation	underlying data structure
	Visitor	operations on a binary tree	2326 B	107	varying operations through visitor	choice of operation
(mixed)	ArithInt	arithmetic-expression interpreter	160 B	59	evaluates any expression	concrete expression
	ChkPt	generic checkpointing routine	2689 B	1343	checkpoints arbitrary object structures	object structure properties
	Image	image-processing framework	4914 B	847	freely composable image filters	specific filter composition
	OoLaLa	matrix computation library	667 B	372	matrices composed from generic objects	specific matrix configuration
	Pipe	function composition	178 B	97	freely composable functions	composition of functions
	Power	power computation	98 B	59	varying operator, exponent, neutral value	specific parameters
	Strategy	image processing	968 B	238	free choice of pixel-wise operator	specific operator

Table II. Benchmark summary.

Compilers normally do not perform aggressive optimizations over primitive-type values, and thus do not perform optimizations similar to those performed by program specialization for these programs.

Object opportunities. The adaptive behavior of some object-oriented programs is controlled completely through object-oriented mechanisms; such programs can be said to provide *pure* object-oriented specialization opportunities. Propagation of type information and simplification of virtual dispatches is sufficient to specialize these programs. In each of our benchmarks, the object composition is fixed in the program, as is all configuration information, meaning that we do not provide the specializer with any information that a compiler could not deduce by statically analyzing the program.

Builder: Matrices with a dense or sparse representation are created using the builder design pattern and subsequently exponentiated. Both matrix creation and matrix multiplication are done independently of the representation. For specialization, the choice of concrete builder is static, and the matrix dimensions and contents are dynamic. After specialization, all operations directly access the concrete matrix representation [Schultz et al. 2000].

Iterator: A set data structure is implemented over an underlying primitive data structure (array or linked list). The iterator design pattern is used to implement the membership and intersection operations independently of the implementation of the underlying data structure. For specialization, the choice of the underlying data structure is static, and the manipulated data is dynamic. After specialization, the counter stored in the iterator object and the values stored in the data structure are accessed directly.

Visitor: Operations are applied to a binary tree of integers, as in Section 5.1. The functions are implemented using the visitor design pattern. New operations can be added without modifying the binary tree implementation. For specialization, the choice of visitors is static, and the tree structure being traversed is dynamic. After specialization, the specialized visitor operations are defined as methods in the tree structure.

Compilers for object-oriented languages are geared towards optimizing these kinds of programs well, and so we expect only limited gains due to program specialization.

Mixed opportunities. In many object-oriented programs, a mixture of object-oriented mechanisms and imperative constructs controls the adaptive behavior. In each program, both primitive-type data and object data in the execution context are static.

ArithInt: A simple arithmetic-expression interpreter is used to compute the maximal value of a function on a given interval. For specialization, the arithmetic expression is static, and the values stored in the environment are dynamic. We specialize the interpreter with respect to a function mapping integer planar coordinates into an integer value. Specializing the interpreter produces a Java arithmetic expression equivalent to the function.

ChkPt: A generic checkpointing routine is used to periodically save the state of a binding-time analysis implemented for a small C-like language. The checkpointing routine can (recursively) checkpoint any object that implements a checkpointing interface. The checkpointing routine can be specialized to object composition properties specific to each phase of the binding-time analysis [Lawall and Muller 2000]. Specialization and benchmarking are done for one phase only. The specialized checkpointing routine only traverses data structures that are modified during this phase. Due to implementation limitations in JSpec, the ChkPt benchmark requires patching after specialization.

Image: A generic image-filtering framework is used to perform standard image-manipulation operations. In this framework, the image representation and the filter to apply are abstracted using design patterns [Schultz et al. 1999]. Complex filtering operations can be performed by composing simpler, basic filter objects such as convolutions and noise reduction. Specialization is done for blurring convolution filters of size 3×3 and 5×5. After specialization, image data is accessed directly, and image filtering is optimized for the chosen convolution matrix.

OoLaLa: The norm1 operation is computed over matrices having a generic representation, as in Section 5.2. The norm1 operation written using iterators is specialized for several concrete representations, namely dense and sparse upper-triangular. After specialization, matrix data is accessed directly, and the matrix iterator is optimized for a specific mode of traversal.

Pipe: A sequence of simple mathematical functions is composed together to form a pipe, and then applied to a single input value. The function composition can be changed freely at run time. For specialization, the function composition is static, and the value input to the function pipe is dynamic. Specialization compiles the arithmetic expression represented by the function into an equivalent Java arithmetic expression.

Power: Exponents are computed, as in the power example from Section 1. The exponent, operator and neutral values are static, and the base value is dynamic. Specialization is done for two objects of class `Power` with different operators.

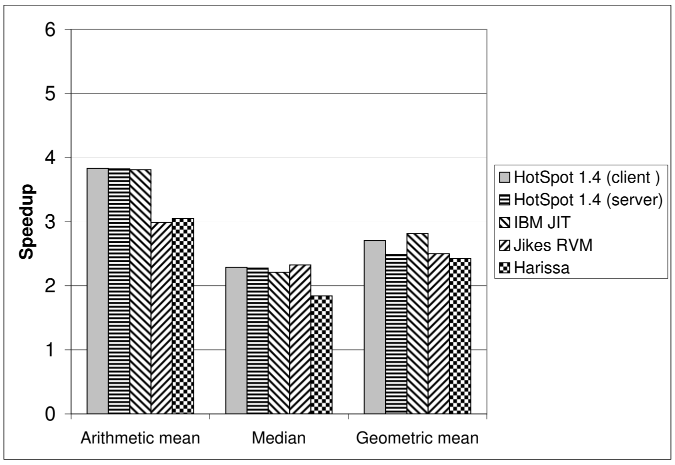
Strategy: A number of single-pixel image operators (e.g., pixel brightness modification) are applied to an image. Each pixel operator is encapsulated into a separate object by using the strategy design pattern [Schultz et al. 2000], so that the choice of image operator can be modified at run time. For specialization, the choice of operators is static, and the image data is dynamic. After specialization, the image operators are applied directly to the image data.

Although standard compilers for object-oriented languages can optimize object interactions in these programs, they do not specialize computations over primitive-type values nor completely reduce megamorphic call points. Thus, we expect significant gains from program specialization.

6.2 Methodology

Experiments are performed on two different machines, a SPARC and an x86. The SPARC machine is a Sun Enterprise 450 running Solaris 2.8, with four 400MHz Ultra-SPARC processors (all benchmarks use only a single thread) and 4Gb of RAM. The x86 machine runs Linux 2.4, and has a single 1.3GHz AMD Athlon processor and 512Mb of RAM.

The aspects resulting from specialization are compiled to Java source code using AspectJ version 0.6b2. We compile Java source code to Java bytecode using Sun's JDK 1.3 `javac` compiler with optimization selected. We compile the resulting bytecode using JIT, adaptive, and off-line compilers: Sun's JDK 1.2.2 JIT compiler for SPARC (referred to as ExactVM) [Sun Microsystems, Inc. 1999], Sun's JDK 1.4.0 HotSpot compiler running in "client" and "server" compilation modes on both SPARC and x86 [Sun Microsystems, Inc. 2002] ("server" compilation mode is more aggressive than the default "client" mode), IBM's JDK 1.3.1 JIT compiler for x86 [IBM 2001], the Jikes Research VM for x86 [IBM 2002], version 2.1.0 with



JVM	Standard deviation
HotSpot 1.4 (client)	3.8
HotSpot 1.4 (server)	5.3
IBM JIT	4.2
Jikes RVM	2.1
Harissa	2.4

Fig. 14. Speedup analysis, x86.

configuration `OptOptSemispace` and optimization level 3 selected[2] (Jikes is the open-source version of the Jalapeño system [Alpern et al. 1999]), and the Harissa off-line bytecode compiler [Muller and Schultz 1999] with optimization level `EO3`[3], using Sun's commercial C compiler version 5.3 on SPARC and gcc version 2.95 on x86. The maximal heap size was set to 120Mb for all systems except Harissa, which does not provide any means of limiting the amount of memory allocated by the program.

Each benchmark program performs ten iterations of the benchmark routine, and discards the first five iterations to allow adaptive compilers to optimize the program. With the adaptive compilers, the first one or two iterations were slower than the remaining iterations, suggesting that dynamic optimizations were being performed during these iterations. All execution times are reported as wall-clock time measured in milliseconds using `java.util.Date`, and the problem size is adjusted to ensure that each main iteration runs long enough to give consistent time measurements, typically between one second and one minute. All benchmarks compute and print a checksum value that is threaded through the computation of each iteration of the benchmark, to prevent compiler optimizations from removing the code that is being benchmarked.

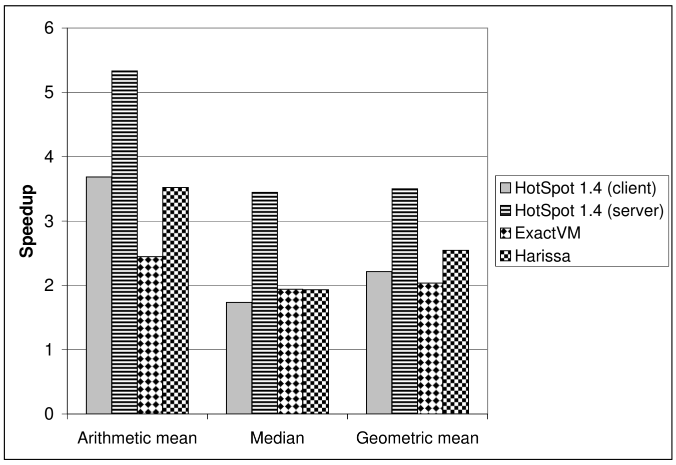
6.3 Results

We now present the benchmark results first by architecture, then by benchmark type, and last individually for each benchmark. The results are assessed in Section 6.4.

Figures 14 (x86) and 15 (SPARC) summarize the overall speedup due to specialization. A value of 2.0 for a given compiler/architecture combination indicates that the specialized programs are on average twice as fast as the generic programs for

[2]For the Strategy benchmark, optimization level 1 was selected to prevent the compiler from crashing.

[3]For the ChkPt benchmark, optimization level `EO1` was selected to limit resource consumption during compilation.



JVM	Standard deviation
HotSpot 1.4 (client)	5.2
HotSpot 1.4 (server)	7.6
ExactVM	1.5
Harissa	3.0

Fig. 15. Speedup analysis, SPARC.

	Geometric mean		Standard deviation	
Benchmark type	x86	SPARC	x86	SPARC
Imperative	3.1	3.2	3.1	4.9
Object-oriented	1.7	1.8	0.4	0.9
Mixed	2.8	2.5	4.7	5.9
Overall	2.6	2.6	3.7	4.7

Table III. Speedup comparison by benchmark type and architecture

this specific compiler/architecture combination. Overall speedups are computed as the arithmetic mean, the median, and the geometric mean of the individual speedups. The geometric mean is included because it is less affected by extreme values than the arithmetic mean. For x86, the geometric mean ranges from 2.4 in the case of Harissa to 2.7 in the case of HotSpot in client mode. For SPARC, the geometric mean ranges from 2.0 in the case of ExactVM to 3.5 in the case of HotSpot in server mode. The standard deviation is reported for each set of values. For both architectures, the standard deviation is relatively high, and is the highest for HotSpot in server mode.

Table III breaks down the overall geometric mean and standard deviation of the speedups by benchmark type. For both architectures, the highest speedups are achieved for the imperative benchmarks, the lowest for the object-oriented benchmarks, and the overall speedup is 2.6 times. The standard deviation is high for both the imperative and mixed benchmarks, and the overall standard deviation is higher for SPARC than for x86.

Figures 16 (x86) and 17 (SPARC) show the speedup due to specialization for each benchmark program. The speedup is computed as the execution time of the generic program divided by the execution time of the corresponding specialized program. Absolute execution times are shown in the Appendix. No results are shown for IBM's JIT on the Image benchmarks, because a runtime error was (incorrectly) generated when running these benchmarks on this system. Similarly, no results are

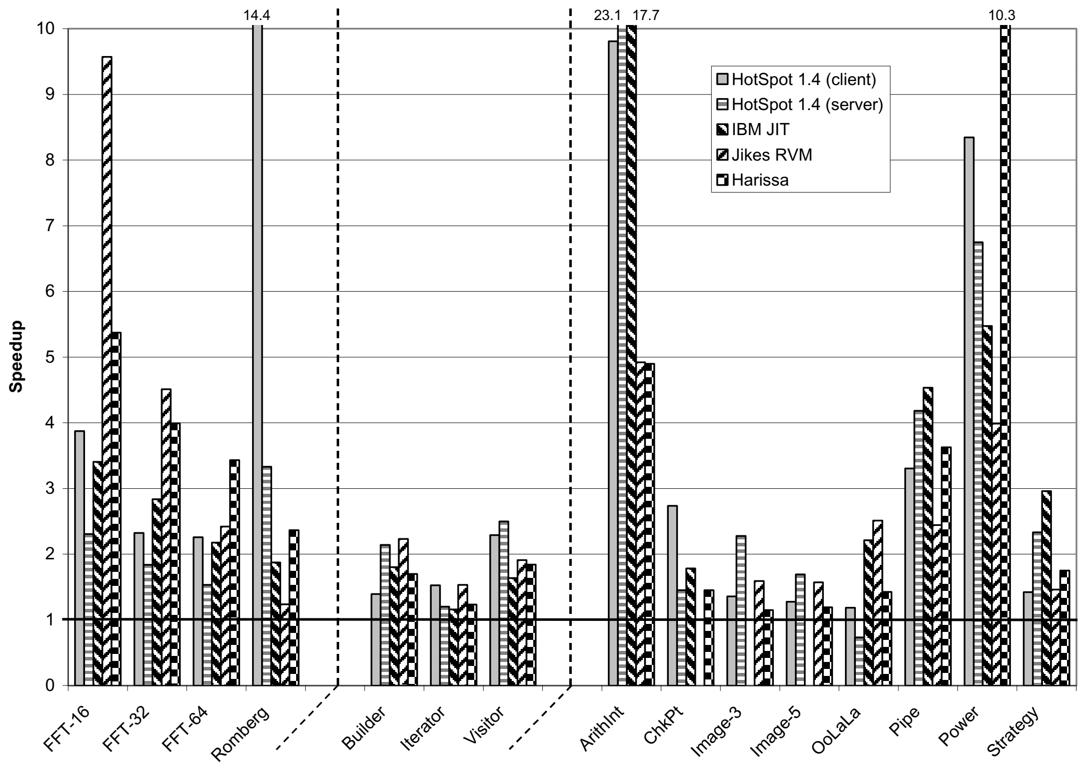


Fig. 16. Detailed speedup comparison (generic time/specialized time), x86.

shown for Jikes on the ChkPt benchmarks, because a stack overflow error occurred, and no means of increasing the stack size could easily be determined.

On both architectures, there are high speedups for the imperative benchmarks FFT (for most variants) and Romberg, and for the mixed benchmarks ArithInt, Power, and Pipe. On SPARC, there are also high speedups for the object-oriented benchmarks Builder and Visitor, and for the mixed benchmark Strategy. Also, on both architectures there are significant speedups for the mixed benchmarks OoLaLa and Image-3, and on x86 there are significant speedups for the object-oriented benchmarks Builder and Visitor as well as the mixed benchmarks ChkPt, Image-3, and Strategy. Slowdowns are observed for OoLaLa on x86 with HotSpot in server mode, and for FFT-64 on SPARC with all systems except HotSpot in server mode.

The standard deviation for each benchmark across compilers is shown in Table IV. The standard deviation is high for Romberg, ArithInt, and Power across both architectures, and relatively high for FFT:16 on x86 and for most variants of FFT on SPARC. When measured across architectures, the standard deviation is bounded by the maximal standard deviation for either architecture, but remains high for FFT:16, Romberg, ArithInt, and Power.

Because Harissa is simply a translator to C code, it can easily be instrumented to generate code that counts the number of virtual calls and class cast checks.

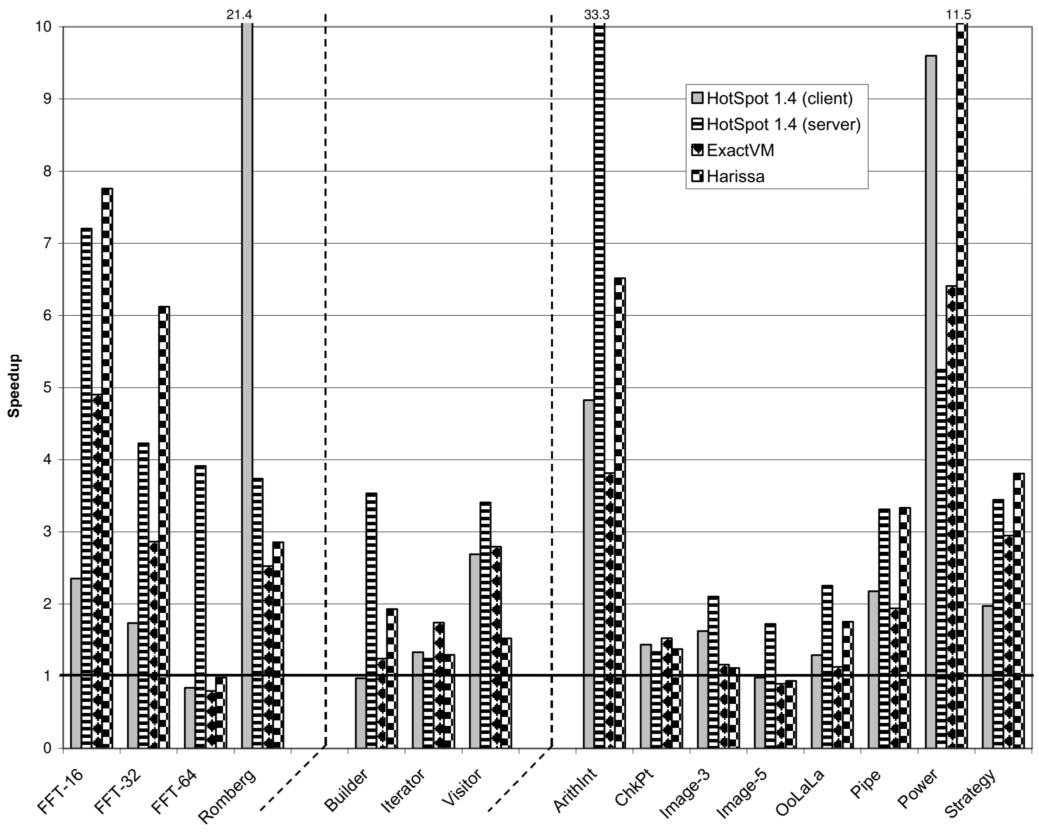


Fig. 17. Detailed speedup comparison (generic time/specialized time), SPARC.

Benchmark	x86	SPARC	Both
FFT:16	2.8	1.1	2.5
FFT:32	1.1	1.9	1.4
FFT:64	0.7	1.5	1.1
Romberg	5.5	9.2	7.0
Builder	0.3	1.1	0.7
Iterator	0.2	0.2	0.2
Visitor	0.4	0.8	0.6
ArithInt	8.1	14.2	10.4
ChkPt	0.6	0.1	0.5
Image:3	0.5	0.5	0.4
Image:5	0.2	0.4	0.3
Pipe	0.8	0.7	0.9
Power	2.5	2.9	2.6
Strategy	0.7	0.8	0.9
OoLaLa	0.7	0.5	0.6

Table IV. Detailed standard deviation comparison

	Virtual calls		Class cast checks	
	generic	specialized	generic	specialized
FFT:16	11	21	0	0
FFT:32	11	21	0	0
FFT:64	11	21	0	0
Romberg	111	111	0	0
Builder	19K	7.7K	0	12K
Iterator	720K	510K	20	210K
Visitor	36K	15K	9	9
ArithInt	111	101	0	0
Image:3	56M	2.8M	56M	110M
Image:5	140M	2.8M	140M	270M
Pipe	181	111	0	30
Power	431	101	0	0
Strategy	2.4M	171	0	10
OoLaLa	160K	2.7K	0	150K

Table V. Comparison of the number of runtime virtual calls and class casts

	Generic	Specialized	Ratio
FFT:16	915	1668	1.8
FFT:32	915	3766	4.1
FFT:64	915	8550	9.3
Romberg	774	1149	1.5
Builder	1463	2140	1.5
Iterator	717	1224	1.7
Visitor	2326	3267	1.4
ArithInt	160	290	1.8
ChkPt	2689	3732	1.4
Image:3	4914	7985	1.6
Image:5	4914	12451	2.5
Pipe	178	341	1.9
Power	98	498	5.1
Strategy	968	1403	1.4
OoLaLa	667	1101	1.7

Table VI. Size of bytecode of targeted program slice

Table V compares the number of virtual calls and class cast checks performed in the targeted program slice when running the generic and specialized versions of each benchmark program using Harissa. The number of virtual dispatches is reduced by the intra-procedural CHA used by Harissa, similarly to the behavior of most standard compilers, whereas the number of class cast checks is not reduced by any Harissa optimizations. As expected, specialization can drastically reduce the number of virtual calls, particularly in the case of Strategy, but can also drastically increase the number of class cast checks, as in Iterator.

With the current implementation of JSpec, specialization always increases the program size, since new methods are added to the program and no methods are removed. Table VI shows the size of the bytecode of the program slice targeted in each benchmark before and after specialization. The size increase due to specialization ranges from 1.4 times in the case of Strategy and Romberg to over 9 in the case of FFT:64.

Inlining is an essential optimization in automatic program specialization for functional and imperative languages (e.g., in program specializers such as Similix [Bondorf 1990] and Tempo [Consel et al. 1996]). In Java, however, inlining across class boundaries cannot easily be performed at the source level due to access modifiers. Nonetheless, our benchmark programs do not use access modifiers, which allows us to examine the effect of source-level inlining on performance. The benchmark results with inlining are shown in the Appendix. These results show that inlining by JSpec provides no advantage in most cases, and thus that the inlining performed by the compilers included in our study in most cases is sufficient. In some cases, inlining is even detrimental to performance.

6.4 Assessment

The geometric mean of the speedups gained by specialization across all compilers is 2.6 on both SPARC and x86, although the standard deviation is higher on SPARC than on x86. As was predicted in Section 6.1, higher speedups are observed on average for the imperative and mixed benchmarks than for the object-oriented benchmarks, although significant speedups are observed for all types of benchmarks. We observe that, with the exception of the ChkPt benchmark, the benchmarks with the lowest speedups are also those with the highest increase in class cast checks (see Table V). Class cast checks are usually relatively expensive operations, and their high number is thus a significant overhead. Thus, we expect that modifying the representation of dynamic data, as mentioned in Section 3.1, will reduce the number of class casts and thus improve the performance of the specialized programs where the number of class casts increases with specialization.

For the OoLaLa benchmark, specialization results in a significant slowdown with Hotspot 1.4.0 in server mode on x86. Given the speedup obtained with IBM's JIT and Jikes, together with the speedup of 1.8 with HotSpot 1.3.1 in server mode (see Appendix for HotSpot 1.3.1 results), we attribute this slowdown to a conflict between specialization and some specific optimization in the HotSpot 1.4.0 server compiler. As for the FFT-64 benchmark on SPARC, slowdowns were observed in all cases except HotSpot 1.4.0 in server mode, where a 3.1 times speedup was observed. Further experimentation is needed to determine the cause of these slowdowns.

The standard deviation for each compiler is relatively high for both architectures, because the speedup that can be gained from specialization depends on the structure of the program, and hence is specific to each benchmark program. The only general exception is the object-oriented benchmarks where a low standard deviation is observed, which we attribute to the fact that smaller speedups are observed overall here than for the imperative and mixed benchmarks. The standard deviation for each individual benchmark is low in most cases across both architectures, but is very high in a few cases, namely FFT:16, Romberg, ArithInt, and Power. Further experimentation is needed to determine the cause of these strong variations in speedup.

Based on our experiments, we observe that specialization significantly optimizes programs beyond the capabilities of the compilers included in this study. Thus, we conclude that program specialization and compiler optimization are complementary, as was argued informally in Section 3.5. Nonetheless, the high variation in speedups also leads us to conclude that when using JSpec to specialize programs, the choice of Java virtual machine may need to be taken into account when deciding how to specialize a program.

7. RELATED WORK

The work presented in this paper relates to other work in program specialization based on partial evaluation, other program specialization techniques for object-oriented languages, compilers for object-oriented languages, and aspect-oriented programming. We discuss each of these in turn.

7.1 Program specialization based on partial evaluation

Run-time specialization for a subset of Java without object-oriented features has been investigated by Masuhara and Yonezawa [Masuhara and Yonezawa 2002], and subsequently extended to include object-oriented features by Affeldt et al. [Affeldt et al. 2002]. Specialization generates bytecode that is contained within a single method encapsulated in a new class, and this class is dynamically loaded by the JVM and compiled using the resident JIT compiler. Their experiments show that JIT compilation of run-time generated code gives equivalent performance to statically generated code, but with a high amortization cost. Compared to JSpec, the most notable differences are that the binding-time analysis is method monovariant, and that the specialized code is residualized using a single method defined in a separate class. As noted in Section 3.2, the use of a method defined in a separate class implies that only public fields can be accessed and that speculative evaluation of virtual dispatches is not possible.

Marquard and Steensgaard developed an on-line partial evaluator for a subset of the object-based language Emerald [Jones et al. 1993; Marquard and Steensgaard 1992; Raj et al. 1991]. Their work focuses on problems particular to on-line specialization. There is no consideration of how partial evaluation should transform an object-oriented program, and virtually no description of how their partial evaluator handles object-oriented features.

Fujinami showed that objects in a C++ program can be specialized at run time based on their state [Fujinami 1998]. The programmer annotates member methods that are to be specialized. Each method is specialized based on the values of selected `private`, `protected`, and `const` fields that are are not modified by the method. Specialization uses standard partial evaluation techniques for imperative languages, and replaces virtual dispatches through static object references by direct method invocations. During specialization, if a virtual call through a static object reference refers to a method that is tagged as `inline`, this method is inlined into the caller method and is itself specialized. This approach to partial evaluation for an object-oriented language concentrates on specializing individual objects. On the contrary, we specialize the interaction that takes place between objects based on their respective state, resulting in a more complete specialization of the program.

Veldhuizen used C++ templates to perform partial evaluation [Veldhuizen 1999]. By combining template parameters and C++ `const` constant declarations, arbitrary computations over primitive values can be performed at compile time. This approach is more limited in its treatment of objects than what we have proposed. For example, objects cannot be dynamically allocated and virtual dispatches cannot be eliminated. Furthermore, the program must be written in a two-level syntax, thus implying that binding-time analysis must be performed manually, and functionality must be implemented twice if both generic and specialized behaviors are needed.

All parts of the specialization process can be done while a program is running, using *dynamic partial evaluation*, as defined by Sullivan in the context of a virtual machine supporting reflection and object-oriented constructs [Sullivan 2001]. With dynamic partial evaluation, specialization is done as a side-effect of normal evaluation. Specialization invariants are manually specified (using a syntax similar to specialization classes), and specialized methods are generated at run-time and automatically selected when appropriate using multi-dispatching. Thus, dynamic partial evaluation can be seen as a cross between traditional partial evaluation and dynamic compiler optimizations. Although potentially more aggressive than standard compilation, dynamic partial evaluation is harder to control than standard partial evaluation.

7.2 Other program specialization techniques

The OoLaLa case study presented in Section 5.2 was carried out in greater detail by Luján et al. using manual specialization techniques [Luján et al. 2001]. A similar study was performed by Budimlić and Kennedy for the OwlPack linear library using an automatic program specialization tool named *JaMake* [Budimlić et al. 1999; Budimlić and Kennedy 2001]. In both cases, the specialized program is compared to an implementation written without the use of object-oriented abstractions, and the overheads due to the use of objects are almost eliminated. Luján et al. characterize the matrix properties and representations for which a set of standard program transformations can eliminate such overheads. The transformations considered by Luján et al. include virtual dispatch elimination and object inlining, but also transformations not normally not included in partial evaluators, such as induction variable elimination. The JaMake tool uses a combination of transformations that essentially derive new classes in which aggressive object inlining has been performed [Budimlić and Kennedy 1999]. Whereas JSpec concentrates on optimizing a program by simplifying its control flow, JaMake concentrates on simplifying its data representation. Hence, these tools are to some extent orthogonal, and both could benefit from incorporating the techniques used in the other.

Java programs can be specialized by transforming the class hierarchy according to a specific usage scenario, as demonstrated by Tip et al. with the tool Jax [Tip et al. 1999]. Given a set of classes that are needed in a specific usage scenario, Jax eliminates unnecessary classes, methods, and fields, merges classes and interfaces when possible, and performs various other operations to reduce the size of the resulting Java program. JSpec on the other hand increases the program size since specialization can increase the size of a method, and specialized methods are simply added to the original program. Nonetheless, specialization also removes static code and can remove dependencies between classes by reducing field access and virtual dispatches, so specialization combined with Jax could give a greater reduction in code size than Jax alone.

7.3 Compilers

The compiler optimization techniques most directly related to our work are customization, selective argument specialization, and concrete type inference; these were compared to program specialization in Section 3.5. Another similar technique is polymorphic inline caching, which allows a compiler to replace a virtual call with an explicit selection between direct calls to the most common callees [Grove et al. 1995; Hölzle and Ungar 1994]. Nevertheless, this optimization must be guided with profile information to avoid code explosion, and retains the cost of a runtime decision.

Sreedhar et al. describe a static analysis that, given a set of Java classes forming a "closed world," partitions the references in the program into two categories: references that only point to objects instantiated from classes in the closed world, and references that may include objects instantiated from classes that have been dynamically loaded [Sreedhar et al. 2000]. This analysis, referred to as *extant analysis*, can be used to automatically determine what side-effects can be performed by dynamically loaded code. JSpec currently requires the programmer to manually specify such side-effects as part of the specialization context, but extant analysis could be used to automate this aspect of the specialization process. The results of the extant analysis could be refined when the program slice targeted by specialization is contained within a *sealed package*. (A sealed package is a Java package stored in a single `jar` file to which no new classes can be added.) Such an approach would be similar to the work by Zaks et al., where the precision of a class hierarchy analysis is improved by taking sealed packages into account [Zaks et al. 2000].

Cooper et al. showed that procedure cloning with respect to a set of values considered to be of interest improves the results of various optimizations by giving a higher degree of polyvariance [Cooper et al. 1992]. A cloned procedure can be optimized for its usage context, but can still be shared between call sites with similar usage contexts, and hence causes less code growth than inlining. Both the analysis and specialization phase of JSpec uses techniques similar to procedure cloning. JSpec uses a polyvariant binding-time analysis that shares method variants for each unique combination of binding times at method call sites. Similarly, during specialization, methods that are specialized for identical contexts are shared between call sites. However, unlike procedure cloning, which limits the number of procedure variants, the number of generated method variants depends on the static information, and thus depends on the static values provided by the programmer.

Automatic object inlining significantly reduces the number of object allocations and operations on fields thereby improving overall runtime performance, as shown by Dolby and Chien [Dolby and Chien 1998; 2000]. Optimizations similar to object inlining are considered future work for JSpec. Nonetheless, since specialization with JSpec simplifies the control flow of the program, we expect that specialization would improve the opportunities for automatic object inlining optimization.

A JIT compiler can be designed as an open-ended framework rather than a closed component, as shown by Ogawa et al. with the OpenJIT system [Ogawa et al. 2000]. The openness of OpenJIT implies that run-time specialization could be added directly into the compilation process. At this level, new methods can be added directly into existing classes, thus circumventing the restrictions of the Java language.

7.4 Aspect-oriented programming

Irwin et al. have shown that sparse matrix code can be implemented efficiently using aspect-oriented programming [Irwin et al. 1997]. Orthogonal issues such as data representation, permutations of rows and columns, and operator fusion are specified in separate aspects, which are then combined by a compiler that generates low-level, optimized code. In the OoLaLa case study presented in Section 5.2, we showed how JSpec can specialize a generic matrix library implementation into a single aspect that contains an implementation optimized for the specified representation. Thus, whereas the aspect-oriented approach is to combine multiple aspects into a single implementation, JSpec combines specialized code generated from multiple classes into a single aspect.

The AspectJ language enables both static and dynamic crosscutting [Kiczales et al. 2001]. Static crosscutting enables method combination (e.g., before, after, around) over existing methods and the introduction of new methods into classes. Dynamic crosscutting enables method combination at programmer-specified points in the dynamic control flow of the program. JSpec uses static crosscutting to introduce specialized methods encapsulated in an aspect into the existing program.

8. CONCLUSION AND FUTURE WORK

The use of frameworks and generic software components is a growing trend in software development; however, this trend comes at the expense of performance. Program specialization can instantiate such frameworks and components for a specific usage context, thus producing efficient implementations. We argue that program specialization is a key technology to reconcile genericness and performance.

In this paper, we have identified overheads intrinsic to generic object-oriented programs. We have demonstrated that such overheads can be automatically reduced by program specialization, and have presented the techniques used to specialize object-oriented programs. To validate our approach, we have developed a program specializer for Java, named JSpec. Using JSpec, we have evaluated the benefits of program specialization by conducting an experimental study on a variety of Java programs. This study has shown not only that JSpec can produce significant speedups (a geometric mean of 2.6 on our benchmarks), but also that JSpec is complementary to optimizing Java compilers.

In future work, we plan to optimize the representation of dynamic data (as outlined in Section 3.1) and to fully support exception handlers in JSpec. In addition, we are considering run-time specialization for the full Java language, but without the restrictions incurred by the approach of Affeldt et al. (see Section 7 for details) [Affeldt et al. 2002]. Nonetheless, changes to our specialization approach are needed to enable run-time specialization while complying with the restrictions of the Java virtual machine.

Availability

JSpec is publicly available at `http://compose.labri.fr/prototypes/jspec/`

REFERENCES

Affeldt, R., Masuhara, H., Sumii, E., and Yonezawa, A. 2002. Supporting objects in run-time bytecode specialization. ACM Press, Aizu, Japan. To appear in the proceedings of Asia-PEPM 2002.

Agesen, O., Palsberg, J., and Schwartzbach, M. 1993. Type inference of SELF. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'93)*. Lecture Notes in Computer Science, vol. 707. Springer-Verlag, Kaiserstautern, Germany, 247–267.

Aigner, G. and Holzle, U. 1996. Eliminating virtual calls in C++ programs. In *Proceedings of ECOOP '96*. Springer-Verlag, Linz, Austria.

Alpern, B., Attanasio, C., Cocchi, A., Lieber, D., Smith, S., Ngo, T., Barton, J., Hummel, S., Sheperd, J., and Mergen, M. 1999. Implementing Jalapeño in Java. See Meissner [1999], 314–324.

Andersen, L. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, Computer Science Department, University of Copenhagen. DIKU Technical Report 94/19.

Baier, R., Glück, R., and Zöchling, R. 1994. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'94)*. Technical Report 94/9, University of Melbourne, Australia, Orlando, FL, USA, 119–132.

Blount, B. and Chatterjee, S. 1999. An evaluation of Java for numerical computing. *Scientific Programming 7(2)*, 97–110. Special Issue: High Performance Java Compilation and Runtime Issues.

Bondorf, A. 1990. Self-applicable partial evaluation. Ph.D. thesis, DIKU, University of Copenhagen, Denmark. Revised version: DIKU Report 90/17.

Braux, M. and Noyé, J. 2000. Towards partially evaluating reflection in Java. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00)*. ACM Press, Boston, MA, USA.

Budimlić, Z. and Kennedy, K. 1999. Prospects for scientific computing in polymorphic, object-oriented style. In *Proceedings of the Ninth SIAM Conference of Parallel Processing in Scientific Computing*. SIAM, San Antonio.

Budimlić, Z. and Kennedy, K. 2001. JaMake: a Java compiler environment. In *Third International Conference on Large Scale Scientific Computing*. Number 2179 in Lecture Notes in Computer Science. Springer-Verlag, Sozopol, Bulgaria, 201–209.

Budimlić, Z., Kennedy, K., and Piper, J. 1999. The cost of being object-oriented: A preliminary study. *Scientific Computing 7,* 2, 87–95.

Chambers, C. and Ungar, D. 1989. Customization: Optimizing compiler technology for SELF, A dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI '89)*, B. Knobe, Ed. ACM Press, Portland, OR, USA, 146–160.

Consel, C. 1993. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*. ACM Press, Copenhagen, Denmark, 66–77.

Consel, C., Hornof, L., Noël, F., Noyé, J., and Volanschi, E. 1996. A uniform approach for compile-time and run-time specialization. In *Partial Evaluation, International Seminar, Dagstuhl Castle*, O. Danvy, R. Glück, and P. Thiemann, Eds. Number 1110 in Lecture Notes in Computer Science. Springer-Verlag, Dagstuhl Castle, Germany, 54–72.

Cooper, K., Hall, M., and Kennedy, K. 1992. Procedure cloning. In *Proceedings of the 1992 International Conference on Computer Languages*. IEEE Computer Society Press, Oakland, CA, USA, 96–105.

Danvy, O. and Filinski, A., Eds. 2001. *Symposium on Programs as Data Objects II*. Lecture Notes in Computer Science, vol. 2053. Aarhus, Denmark.

Dean, J., Chambers, C., and Grove, D. 1995. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*. ACM SIGPLAN Notices, 30(6), La Jolla, CA USA, 93–102.

Dean, J., Grove, D., and Chambers, C. 1995a. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95)*, W. G. Olthoff, Ed. Lecture Notes in Computer Science, vol. 952. Springer-Verlag, Århus, Denmark, 77–101.

Dean, J., Grove, D., and Chambers, C. 1995b. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP '95*. Springer-Verlag, Aarhus, Denmark.

Detlefs, D. and Agesen, O. 1999. Inlining of virtual methods. See Guerraoui [1999], 258–278.

Dolby, J. and Chien, A. 1998. An evaluation of automatic object inline allocation techniques. In *OOPSLA'97 Conference Proceedings*. ACM SIGPLAN Notices. ACM Press, ACM Press, Vancouver, Canada, 1–20.

Dolby, J. and Chien, A. 2000. An automatic object inlining optimizations and its evaluation. See Lam [2000], 345–357.

Fujinami, N. 1998. Determination of dynamic method dispatches using run-time code generation. In *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, X. Leroy and A. Ohori, Eds. Lecture Notes in Computer Science, vol. 1473. Springer-Verlag, Kyoto, Japan, 253–271.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Grove, D., Dean, J., Garrett, C., and Chambers, C. 1995. Profile-guided receiver class prediction. In *OOPSLA'95 Conference Proceedings*. ACM Press, Austin, TX, USA, 108–123.

Guerraoui, R., Ed. 1999. *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*. Lecture Notes in Computer Science, vol. 1628. Springer-Verlag, Lisbon, Portugal.

Hölzle, U. and Ungar, D. 1994. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI'94)*. ACM SIGPLAN Notices, 29(6), New York, NY, USA, 326–336.

Hornof, L. and Noyé, J. 2000. Accurate binding-time analysis for imperative languages: Flow, context, and return sensitivity. *Theoretical Computer Science (TCS) 248,* 1–2, 3–27.

Hornof, L., Noyé, J., and Consel, C. 1997. Effective specialization of realistic programs via use sensitivity. In *Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, P. Van Hentenryck, Ed. Lecture Notes in Computer Science, vol. 1302. Springer-Verlag, Paris, France, 293–314.

IBM. 2001. IBM JDK 1.3.1. Accessible from `http://www.ibm.com/java/jdk`.

IBM. 2002. Jikes RVM 2.1.0. Accessible from `http://www.ibm.com/developerworks/oss/jikesrvm/`.

Irwin, J., Loingtier, J., Gilbert, J., Kiczales, G., Lamping, J., Mendhekar, A., and Shpeisman, T. 1997. Aspect-oriented programming of sparse matrix code. In *Proceedings of the First International Conference on Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Y. Ishikawa, R. Oldehoeft, J. Reynders, and M. Tholsburn, Eds. Lecture Notes in Computer Science, vol. 1343. Springer-Verlag, Marina del Rey, CA, USA.

Ishizaki, K., Kawahito, M., Yasue, T., Komatsu, H., and Nokatani, T. 2000. A study of devirtualization techniques for a Java Just-In-Time compiler. See Rosson and Lea [2000], 294–310.

Java Grande Forum. 1999. The Java Grande Forum benchmark suite. Accessible from `http://www.javagrande.org`.

Jones, N., Gomard, C., and Sestoft, P. 1993. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. 2001. An overview of AspectJ. See Knudsen [2001], 327–353.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. 1997. Aspect-oriented programming. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'97)*, M. Aksit and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 1241. Springer, Jyväskylä, Finland, 220–242.

Knudsen, J., Ed. 2001. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01)*. Lecture Notes in Computer Science, vol. 2072. Budapest, Hungary.

LAM, M., Ed. 2000. *Proceedings of the ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00).* Vancouver, British Columbia, Canada.

LAWALL, J. AND MULLER, G. 2000. Efficient incremental checkpointing of Java programs. In *Proceedings of the International Conference on Dependable Systems and Networks.* IEEE, New York, NY, USA, 61–70.

LLOYD, J. AND SHEPHERDSON, J. 1991. Partial evaluation in logic programming. *Journal of Logic Programming 11*, 217–242.

LUJÁN, M. 1999. Object oriented linear algebra. M.S. thesis, University of Manchester.

LUJÁN, M., FREEMAN, T., AND GURD, J. 2000. OOLALA: an object oriented analysis and design of numerical linear algebra. See Rosson and Lea [2000], 229–252.

LUJÁN, M., GURD, J., AND FREEMAN, T. 2001. OOLALA: Transformations for implementations of matrix operations at high abstraction levels. In *Proceedings 4th Workshop on Parallel Object-Oriented Scientific Computing — POOSC'01.* ACM Press, Tampa Bay, Florida, USA.

MARQUARD, M. AND STEENSGAARD, B. 1992. Partial evaluation of an object-oriented imperative language. M.S. thesis, University of Copenhagen.

MASUHARA, H. AND YONEZAWA, A. 2002. A portable approach to dynamic optimization in run-time specialization. *New Generation Computing 20,* 1, 101–124.

MEISSNER, L., Ed. 1999. *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99).* ACM SIGPLAN Notices, vol. 34(10). ACM Press, Denver, Colorado, USA.

MULLER, G. AND SCHULTZ, U. 1999. Harissa: A hybrid approach to Java execution. *IEEE Software 16,* 2 (Mar.), 44–51.

OGAWA, H., SHIMURA, K., MATSUOKA, S., MARUYAMA, F., SOHDA, Y., AND KIMURA, Y. 2000. OpenJIT: an open-ended, reflective JIT compiler framework for Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'00).* Lecture Notes in Computer Science, vol. 1850. Springer-Verlag, Cannes, France.

PLEVYAK, J. AND CHIEN, A. 1994. Precise concrete type inference for object-oriented languages. In *OOPSLA'94 Conference Proceedings.* SIGPLAN Notices, vol. 29:10. ACM Press, ACM Press, Portland, OR, USA, 324–324.

RAJ, R., TEMPERO, E., LEVY, H., BLACK, A., HUTCHINSON, N., AND JUL, E. 1991. Emerald: A general-purpose programming language. *Software — Practice and Experience 21,* 1 (Jan.), 91–118.

ROSSON, M. AND LEA, D., Eds. 2000. *OOPSLA'00 Conference Proceedings.* ACM SIGPLAN Notices. ACM Press, ACM Press, Minneapolis, MN USA.

SCHULTZ, U. 2000. Object-oriented software engineering using partial evaluation. Ph.D. thesis, University of Rennes I, Rennes, France.

SCHULTZ, U. 2001. Partial evaluation for class-based object-oriented languages. See Danvy and Filinski [2001], 173–197.

SCHULTZ, U., LAWALL, J., CONSEL, C., AND MULLER, G. 1999. Towards automatic specialization of Java programs. See Guerraoui [1999], 367–390.

SCHULTZ, U., LAWALL, J., CONSEL, C., AND MULLER, G. 2000. Specialization patterns. In *Proceedings of the* 15th *IEEE International Conference on Automated Software Engineering (ASE 2000).* IEEE Computer Society Press, Grenoble, France, 197–206.

SPEC. 1998. SPEC JVM 98 benchmarks. Standard Performance Evaluation Corporation. Accessible from `http://www.specbench.org/osg/jvm98/`.

SREEDHAR, V., BURKE, M., AND CHOI, J. 2000. A framework for interprocedural optimization in the presence of dynamic class loading. See Lam [2000], 196–207.

SULLIVAN, G. 2001. Dynamic partial evaluation. See Danvy and Filinski [2001], 238–256.

SUN MICROSYSTEMS, INC. 1999. Sun JDK 1.2.2. Accessible from `http://java.sun.com/products/j2se`.

SUN MICROSYSTEMS, INC. 2002. Sun JDK 1.4.0. Accessible from `http://java.sun.com/products/j2se`.

SUNDARESAN, V., HENDREN, L., AND RAZAFIMAHEFA, C. 2000. Practical virtual method call resolution for Java. See Rosson and Lea [2000], 264–280.

Name. The name of the associated design pattern.

Description. A short description of the design pattern.

Extent. The program slice that is relevant when optimizing a use of the design pattern.

Overhead. Possible overheads associated with use of the design pattern.

Compiler. Analysis of when these overheads are eliminated by standard compilers.

Approach. Specialization strategies that eliminate the identified overheads.

Condition. The conditions under which the specialization strategies can be effectively exploited.

Specialization class. Guidelines for how to write the needed specialization classes, and how to most effectively apply them.

Applicability. A rating of the overall applicability of specialization to a use of the design pattern, using the other information categories as criteria.

Example. An example of the use of specialization to eliminate the identified overhead; the example may include specialization classes or textual descriptions.

Fig. 18. Specialization pattern template

Tip, F., Laffra, C., and Sweeney, P. 1999. Practical experience with an application extractor for Java. See Meissner [1999], 292–305.

Veldhuizen, T. 1999. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'98).* ACM Press, San Antonio, TX, USA, 13–18.

Veldhuizen, T. 2000. Expression templates in Java. In *Generative and Component-Based Software Engineering, Second International Symposium, GCSE'00*, G. Butler and S. Jarzabek, Eds. Lecture Notes in Computer Science, vol. 2177. Springer, Erfurt, Germany, 188–202. Revised Papers.

Volanschi, E., Consel, C., Muller, G., and Cowan, C. 1997. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings.* ACM Press, Atlanta, GA, USA, 286–300.

Wang, T. and Smith, S. 2001. Precise constraint-based type inference for Java. See Knudsen [2001], 99–117.

XEROX 2000. AspectJ home page. Accessible from `http://aspectj.org`. Xerox Corp.

Zaks, A., Feldman, V., and Aizikowitz, N. 2000. Sealed calls in Java packages. See Rosson and Lea [2000], 83–92.

A. SPECIALIZATION PATTERNS

This appendix describes two specialization patterns, the visitor specialization pattern and the iterator specialization pattern. A specialization pattern describes the overheads intrinsic in using a particular design pattern, and documents how to use specialization to eliminate these overheads [Schultz et al. 2000]. In addition, a specialization pattern can refer to other specialization patterns, to describe how multiple design patterns can be specialized together. Specialization patterns not only guide specialization after a program has been written, but can also help the programmer structure the program so that subsequent specialization will be beneficial.

We use the standard specialization pattern template shown in Figure 18, with a brief entry for each item; this level of detail allows us to concisely convey the essence of each specialization pattern.

A.1 Visitor specialization pattern

Name. Visitor pattern.

Description. The visitor pattern allows an operation to be performed on the elements of an object structure to be specified externally to the classes that define this structure. Each element of the object structure defines an `accept` method that calls a matching method in the visitor.

Extent. Specialization is applied to the concrete visitors and the object structure that they traverse.

Overhead. Double-dispatching through the `accept` methods is used to select visitor methods based on the concrete type of the object that is currently being visited.

Compiler. When only a single visitor is used, a compiler can normally eliminate the double dispatching. When multiple visitors are used, a compiler normally cannot predict the program control flow, and hence cannot eliminate the double dispatching.

Approach. By specializing the object structure for the visitor argument, speculative specialization can generate specialized versions of each accept method, where the visitor methods are directly invoked. Moreover, if the object structure is known, all traversal can be completely eliminated.

Condition. If the choice of visitor is fixed, and the visitor argument is passed directly through the `accept` methods, then the object structure can be specialized to the visitor. If the object structure is invariant, and is accessed in a fixed way, then all traversal can be specialized away.

Specialization class. The specialization class should fix the type of the visitor. If the object structure is known, the specialization class should also declare this fact.

Applicability. High when the object structure is invariant, medium when only the visitor is known but the amount of work done in each node is limited, low otherwise.

Example. See Section 5.1.

A.2 Iterator specialization pattern

Name. Iterator pattern.

Description. The iterator pattern allows the traversal of a structure to be expressed in terms of a standard interface; a client need only know the iterator interface to enumerate the elements of a structure.

Extent. Specialization is applied to the iterator and the structure on which it operates.

Overhead. Decoupling of the client from the structure that it manipulates.

Compiler. When multiple structures are manipulated through different iterators, a compiler normally cannot resolve the virtual calls used to access structure elements.

Approach. A client that is given an iterator can be specialized to the type of this iterator and optionally to its state. A client that accesses a given structure through an iterator obtained from the structure can be specialized to the type of this structure and optionally to its state. Specialization will in both cases make access to the underlying structure explicit, and may simplify each iteration step.

Name	T_{Gen}	T_{Spec}	SF	$T_{\text{S+I}}$	SF$_{+I}$
ArithInt	8.34	0.85	9.8	1.08	7.7
Pipe	2.98	0.9	3.3	0.88	3.4
Power	11.02	1.32	8.3	1.37	8.1
Romberg	48.39	3.37	14.4	N/A	N/A
Builder	4.62	3.32	1.4	3.52	1.3
Iterator	7.56	4.95	1.5	5.74	1.3
Strategy	2.29	1.61	1.4	1.3	1.8
Visitor	4.12	1.8	2.3	N/A	N/A
FFT-16	2.87	0.93	3.1	N/A	N/A
FFT-32	5.56	2.4	2.3	N/A	N/A
FFT-64	11.49	5.09	2.3	N/A	N/A
Image-3	1.02	0.75	1.4	0.67	1.5
Image-5	2.2	1.73	1.3	22.51	0.1
OoLaLa	17.86	15.07	1.2	18.0	1.0
ChkPt	13.86	6.68	2.1	N/A	N/A
Average			3.7		3.5

Table VII. Results for x86 HotSpot 1.4 (client)

Name	T_{Gen}	T_{Spec}	SF	$T_{\text{S+I}}$	SF$_{+I}$
ArithInt	4.0	0.17	23.1	0.18	22.6
Pipe	1.93	0.46	4.2	0.46	4.2
Power	4.22	0.63	6.7	0.62	6.8
Romberg	4.71	1.42	3.3	N/A	N/A
Builder	2.92	1.36	2.1	0.86	3.4
Iterator	3.19	2.66	1.2	2.27	1.4
Strategy	1.81	0.78	2.3	0.76	2.4
Visitor	3.05	1.22	2.5	N/A	N/A
FFT-16	3.48	1.51	2.3	N/A	N/A
FFT-32	6.04	3.28	1.8	N/A	N/A
FFT-64	11.01	7.18	1.5	N/A	N/A
Image-3	1.45	0.64	2.3	0.61	2.4
Image-5	1.93	1.14	1.7	21.85	0.1
OoLaLa	11.92	16.26	0.7	16.92	0.7
ChkPt	8.11	5.59	1.5	N/A	N/A
Average			3.8		3.8

Table VIII. Results for x86 HotSpot 1.4 (server)

Condition. The iterator or the structure that is iterated must be known, and the iterator must be manipulated in a fixed way throughout the parts of the client that access the structure.

Specialization class. The specialization class should either fix the type of the iterator of fix the type of the underlying structure.

Applicability. High when many elements are used, low when the traversal algorithm is complex and cannot be simplified by specialization, medium otherwise.

Example. See Section 5.2.

B. DETAILED BENCHMARK RESULTS

Tables VII through XVI present the execution times and speedup factors for the benchmarks presented in Section 6. Results for x86 are presented first, followed by results for SPARC. In each table, the column T_{Gen} gives the execution time of the generic version of the program, the column T_{Spec} gives the execution time of the specialized version of the program, the column SF gives the speedup factor of the specialized version over the generic version (obtained by dividing the execution of the former by that of the latter), the column $T_{\text{S+I}}$ gives the execution time of the specialized, inlined version of the program (no value in this column indicates that the specialized code does not contain any function calls, and thus inlining is irrelevant), and the column SF$_{+I}$ gives the speedup factor of the specialized, inlined version over the generic version (obtained as for SF). Times are measured in seconds. Averages are computed using the arithmetic mean.

The results for Sun's JDK 1.3.1 HotSpot compiler running in server mode for x86 are included in Table XII; these results are not included in the general discussion of Section 6 but are compared to the results of Sun's JDK 1.4.0 HotSpot compiler in Section 6.4.

Name	T_{Gen}	T_{Spec}	SF	$T_{\text{S+I}}$	SF_{+I}
ArithInt	5.43	0.31	17.7	0.39	14.0
Pipe	1.54	0.34	4.5	0.34	4.5
Power	3.2	0.63	5.0	0.67	4.8
Romberg	3.89	2.08	1.9	N/A	N/A
Builder	3.96	2.2	1.8	2.15	1.8
Iterator	3.23	2.79	1.2	2.52	1.3
Strategy	1.31	0.63	2.1	0.62	2.1
Visitor	2.15	1.32	1.6	N/A	N/A
FFT-16	1.34	0.39	3.4	N/A	N/A
FFT-32	2.48	0.87	2.8	N/A	N/A
FFT-64	4.66	2.14	2.2	N/A	N/A
Image-3	0.71	N/A	N/A	N/A	N/A
Image-5	1.41	N/A	N/A	N/A	N/A
OoLaLa	6.22	2.81	2.2	2.14	2.9
ChkPt	33.25	18.67	1.8	N/A	N/A
Average			3.7		3.5

Table IX. Results for x86 IBM JIT

Name	T_{Gen}	T_{Spec}	SF	$T_{\text{S+I}}$	SF_{+I}
ArithInt	6.5	1.32	4.9	0.82	8.0
Pipe	1.93	0.79	2.4	0.7	2.8
Power	4.11	1.03	4.0	0.95	4.3
Romberg	9.37	7.59	1.2	N/A	N/A
Builder	3.78	1.69	2.2	1.63	2.3
Iterator	4.62	3.02	1.5	2.96	1.6
Strategy	2.03	1.39	1.5	1.44	1.4
Visitor	2.69	1.41	1.9	N/A	N/A
FFT-16	5.25	0.55	9.6	N/A	N/A
FFT-32	7.46	1.66	4.5	N/A	N/A
FFT-64	11.19	4.62	2.4	N/A	N/A
Image-3	0.93	0.59	1.6	0.56	1.7
Image-5	1.99	1.27	1.6	1.45	1.4
OoLaLa	9.36	3.73	2.5	4.18	2.2
ChkPt	N/A	N/A	N/A	N/A	N/A
Average			3.0		3.2

Table X. Results for x86 Jikes RVM

Name	T_{Gen}	T_{Spec}	SF	$T_{\text{S+I}}$	SF_{+I}
ArithInt	17.84	3.64	4.9	2.26	7.9
Pipe	4.53	1.25	3.6	1.46	3.1
Power	6.21	0.6	10.3	1.13	5.5
Romberg	2.71	1.15	2.4	N/A	N/A
Builder	3.93	3.68	1.1	3.28	1.2
Iterator	8.22	6.67	1.2	7.05	1.2
Strategy	1.64	0.94	1.8	0.98	1.7
Visitor	2.96	1.61	1.8	N/A	N/A
FFT-16	1.45	0.27	5.4	N/A	N/A
FFT-32	2.77	0.69	4.0	N/A	N/A
FFT-64	5.44	1.59	3.4	N/A	N/A
Image-3	2.05	1.78	1.1	1.29	1.6
Image-5	3.98	3.34	1.2	1.77	2.2
OoLaLa	13.3	9.34	1.4	9.85	1.3
ChkPt	15.97	11.0	1.5	N/A	N/A
Average			3.0		2.9

Table XI. Results for x86 Harissa

Name	T_{Gen}	T_{Spec}	SF	$T_{\text{S+I}}$	SF_{+I}
ArithInt	6.01	0.39	15.6	0.39	15.6
Pipe	1.89	0.54	3.5	0.43	4.4
Power	3.99	0.62	6.4	0.63	6.4
Romberg	4.58	1.42	3.2	N/A	N/A
Builder	3.06	1.39	2.2	1.09	2.8
Iterator	5.09	2.83	1.8	2.64	1.9
Strategy	1.78	0.91	2.0	0.76	2.3
Visitor	3.12	1.26	2.5	N/A	N/A
FFT-16	4.39	1.49	3.0	N/A	N/A
FFT-32	7.24	3.27	2.2	N/A	N/A
FFT-64	12.34	7.1	1.7	N/A	N/A
Image-3	0.91	0.58	1.6	0.53	1.7
Image-5	1.99	1.12	1.8	21.84	0.1
OoLaLa	12.48	6.88	1.8	6.53	1.9
ChkPt	8.67	6.03	1.4	N/A	N/A
Average			3.4		3.4

Table XII. Results for x86 HotSpot 1.3 (server)

Received. . .

Name	T_{Gen}	T_{Spec}	SF	$T_{\text{S+I}}$	SF_{+I}
ArithInt	21.29	4.41	4.8	4.77	4.5
Pipe	7.77	3.57	2.2	3.85	2.0
Power	30.7	3.2	9.6	3.23	9.5
Romberg	146.28	6.83	21.4	N/A	N/A
Builder	14.35	14.77	1.0	14.65	1.0
Iterator	29.25	21.95	1.3	24.52	1.2
Strategy	4.31	2.18	2.0	2.09	2.1
Visitor	15.02	5.59	2.7	N/A	N/A
FFT-16	7.26	3.09	2.4	N/A	N/A
FFT-32	14.13	8.15	1.7	N/A	N/A
FFT-64	27.77	33.08	0.8	N/A	N/A
Image-3	3.18	3.0	1.1	2.36	1.4
Image-5	7.35	7.5	1.0	64.21	0.1
OoLaLa	57.76	44.71	1.3	38.5	1.5
ChkPt	37.39	26.04	1.4	N/A	N/A
Average			3.6		3.6

Table XIII. Results for SPARC HotSpot 1.4 (client)

Name	T_{Gen}	T_{Spec}	SF	$T_{\text{S+I}}$	SF_{+I}
ArithInt	15.99	0.48	33.3	2.06	7.7
Pipe	7.54	2.28	3.3	2.26	3.3
Power	15.62	2.98	5.2	3.27	4.8
Romberg	11.31	3.03	3.7	N/A	N/A
Builder	13.59	3.85	3.5	4.0	3.4
Iterator	14.94	12.01	1.2	10.85	1.4
Strategy	3.49	1.01	3.4	1.02	3.4
Visitor	14.11	4.14	3.4	N/A	N/A
FFT-16	5.22	0.73	7.2	N/A	N/A
FFT-32	8.09	2.01	4.0	N/A	N/A
FFT-64	13.48	4.36	3.1	N/A	N/A
Image-3	2.17	1.08	2.0	0.89	2.4
Image-5	5.05	2.93	1.7	65.59	0.1
OoLaLa	47.84	21.23	2.3	19.98	2.4
ChkPt	28.03	21.0	1.3	N/A	N/A
Average			5.2		3.4

Table XIV. Results for SPARC HotSpot 1.4 (server)

Name	T_{Gen}	T_{Spec}	SF	$T_{\text{S+I}}$	SF_{+I}
ArithInt	19.51	5.11	3.8	3.93	5.0
Pipe	7.45	3.84	1.9	3.49	2.1
Power	21.73	3.39	6.4	3.07	7.1
Romberg	15.63	6.19	2.5	N/A	N/A
Builder	12.96	10.44	1.2	11.7	1.1
Iterator	23.64	22.0	1.1	20.33	1.2
Strategy	4.82	2.3	2.1	2.44	2.0
Visitor	13.45	4.81	2.8	N/A	N/A
FFT-16	9.5	1.94	4.9	N/A	N/A
FFT-32	16.31	5.69	2.9	N/A	N/A
FFT-64	29.27	36.77	0.8	N/A	N/A
Image-3	3.35	2.89	1.2	2.26	1.5
Image-5	7.93	8.88	0.9	7.29	1.1
OoLaLa	48.88	43.26	1.1	35.44	1.4
ChkPt	66.15	43.37	1.5	N/A	N/A
Average			2.3		2.5

Table XV. Results for SPARC ExactVM

Name	T_{Gen}	T_{Spec}	SF	$T_{\text{S+I}}$	SF_{+I}
ArithInt	21.73	3.34	6.5	3.1	7.0
Pipe	6.63	1.99	3.3	1.95	3.4
Power	23.54	2.05	11.5	2.09	11.3
Romberg	9.47	3.32	2.9	N/A	N/A
Builder	15.39	7.97	1.9	7.64	2.0
Iterator	30.59	23.65	1.3	23.86	1.3
Strategy	3.15	0.83	3.8	0.82	3.8
Visitor	8.11	5.32	1.5	N/A	N/A
FFT-16	4.07	0.52	7.8	N/A	N/A
FFT-32	6.78	1.11	6.1	N/A	N/A
FFT-64	12.58	12.82	1.0	N/A	N/A
Image-3	6.99	6.27	1.1	3.96	1.8
Image-5	12.95	13.88	0.9	9.3	1.4
OoLaLa	60.6	34.52	1.8	37.23	1.6
ChkPt	16.4	11.92	1.4	N/A	N/A
Average			3.5		3.6

Table XVI. Results for SPARC Harissa