Towards Automatic Specialization of Java Programs[⋆]

Ulrik Pagh Schultz, Julia L. Lawall[⋆⋆], Charles Consel, and Gilles Muller

Compose Group, IRISA
Campus Universitaire de Beaulieu, F-35042 Rennes Cedex, France
{ups,jll,consel,muller}@irisa.fr
http://www.irisa.fr/compose

Abstract. Automatic program specialization can derive efficient implementations from generic components, thus reconciling the often opposing goals of genericity and efficiency. This technique has proved useful within the domains of imperative, functional, and logical languages, but so far has not been explored within the domain of object-oriented languages. We present experiments in the specialization of Java programs. We demonstrate how to construct a program specializer for Java programs from an existing specializer for C programs and a Java-to-C compiler. Specialization is managed using a declarative approach that abstracts over the optimization process and masks implementation details. Our experiments show that program specialization provides a four-time speedup of an image-filtering program. Based on these experiments, we identify optimizations of object-oriented programs that can be carried out by automatic program specialization. We argue that program specialization is useful in the field of software components, allowing a generic component to be specialized to a specific configuration.

[⋆] Supported in part by BULL, and in part by Alcatel under the Reutel 2000 contract.
[⋆⋆] Supported in part by an H.H. Powers grant from Oberlin College, and by NSF grant EIA-9806718. Author's current address: Department of Computer Science, Brandeis University, Waltham, MA 02254. e-mail: jll@cs.brandeis.edu

1 Introduction

Although initially designed for embedded systems and marketed as a language for web-based programming, Java is increasingly gaining acceptance as a general-purpose language. The object-oriented paradigm has well-recognized advantages for application design, and more specifically for program structuring. It makes it possible to decompose an application into well-defined, generic components, closely corresponding to the structure of the modeled problem. This structuring leads to a number of important software engineering improvements regarding maintainability and re-usability of code. However, these advantages often translate into a loss in performance.

The conflict between software generality and performance has been recognized in areas such as operating systems [6] and graphics [20]. This conflict is being increasingly addressed, with success, using forms of *program specialization* [21]. Program specialization adapts a generic program component to a given usage context. This approach can lead to considerable performance gains, by eliminating from the general code all aspects not related to that precise context.

Specialization has been performed manually by adapting critical program components to the most common usage patterns [5, 28, 29]. Manual specialization improves performance, but has a limited applicability, because the process is error-prone. Recently, tools have been developed to automatically specialize programs [1–3, 7, 8, 18, 19]. Applications of program specialization are emerging in a number of fields, including scientific code [3, 4, 8], systems software [13, 24], and computer graphics [15], with very promising results. However, automatic specialization of object-oriented programs remains uninvestigated. Given the existing base of experience with imperative languages, we construct a specializer to optimize object-oriented programs, specifically programs written in Java.

Developing an automatic specializer for a realistic imperative language is a long and complex task. Rather than designing a Java specializer from scratch, we are experimenting with an approach based on the Tempo [8] specializer for C programs, coupled with the Harissa optimizing Java-to-C compiler [25, 26].

In this paper, we present preliminary experiments in specializing Java programs. Our contributions are as follows:

– We identify specialization transformations that are particularly useful in object-oriented languages. These transformations are implemented within a single, uniform framework.
– We illustrate our approach with a concrete example, a graphical filtering application, transforming the generic code into a form close to hand-optimized code.
– We apply the declarative approach to specialization proposed by Volanschi *et al.* [32] to specify specialization strategies in a high-level manner. In contrast to Volanschi *et al.*'s work, however, we perform specialization automatically, not manually.
– We demonstrate that the choice of C as a target language for specializing Java programs makes it possible to address optimizations at all levels from

the source program to the run-time environment. We also address the issue of expressing the result of program specialization as Java source code.
– We argue that program specialization is a key approach to improving the performance of generic software components, by adapting the code of a component to its configuration.

This paper is organized as follows: first, Section 2 informally presents specialization. Then, Section 3 introduces our example program. This program is specialized in Section 4, where we identify some typical opportunities for specialization in Java programs. Afterwards, Section 5 describes the functionality of the staged program specialization process implemented by our prototype. Section 6 describes related work. Section 7 discusses future work. Finally, Section 8 concludes.

2 Background

Program specialization has been explored for a variety of languages ranging from functional to logic languages. In recent years, program specializers for real-sized languages such as Fortran [3] and C [1, 8] have been developed. Realistic applications of program specializers to areas such as systems software [24] and scientific computing [3] have clearly demonstrated that automatic program specialization is an effective tool to allow programmers to write *generic* programs without loss of efficiency.

Although object-oriented languages encourage genericity, and thus offer opportunities for specialization, specialization has so far been mostly unexplored for this class of languages.

In this section, program specialization is introduced. We also present a declarative approach to specifying how a program should be specialized.

2.1 Program Specialization

Intuitively, program specialization is aimed at instantiating a program with respect to some of its parameters. By restricting a generic program to a specific usage context, one hopes to enable aggressive optimizations. For an object-oriented program, this approach can be applied to the methods of a class. A method can be specialized with respect to parts of its calling context including parameters, fields of the enclosing object, and static fields of other classes. For example, the parameters of a method may represent options to be analyzed to determine a particular task to be performed. Specialization can typically eliminate the interpretation of such contextual information.

One approach to program specialization is *partial evaluation*, which performs aggressive interprocedural constant propagation (of all data types). Based on the input values specified by the user and on constants explicit in the program, partial evaluation does constant folding, as well as optimizations such as loop unrolling and some forms of strength reduction. Computations that can be simplified based on information available during specialization are known as *static.*

```
class Power {
  private int exp;
  Power( int e ) { this.exp = e; }
  int calculate( int b ) {
    int res = 1;
    for( int i=0; i<this.exp; i++ )
      res *= b;
    return res;
  }
}
```

Fig. 1. A Java class for the power function.

Other computations are known as *dynamic*, and are reconstructed literally to form the specialized program.

Let us illustrate program specialization with a simple Java class for computing the power function, displayed in Figure 1.

Assume the `calculate` method is invoked repeatedly with a specific exponent, say `3`, within a loop where only the base changes. In this situation, it is worthwhile to specialize the `calculate` method with respect to the given exponent. Specialization unrolls the loop in the `calculate` method, producing the following specialized method.

```
int calculate_3( int b ) {
  int res;
  res = b * b * b;
  return res;
}
```

The set of values to specialize over may become gradually available during compilation and execution. Thus, a program may need to be specialized both at compile time and at run time. In the `Power` class example, we could have dynamically generated a specialized version of `calculate` for any exponent supplied at run time. In fact, the partial evaluator Tempo offers both strategies in a uniform environment [9]. Run-time specialization widens the opportunities to eliminate genericity in programs.

Partial evaluation differs from ordinary optimization in that no resource limits are imposed on the computations that are performed during transformation. While this enables transformations that are out of the scope of an ordinary compiler, it does imply that the partial evaluation process must be guided by the user. Because the process of specializing parts of a large program, and then reorganizing the program to use the specialized code, is complex, we need a language for declaring specialization opportunities.

2.2 A Declarative Approach to Program Specialization

Volanschi *et al.* [32] present a declarative approach for specifying specialization opportunities in object-oriented programs. Specialization opportunities are declared separately from the program, in the form of *specialization classes*. A specialization class defines the conditions under which specialization should occur and what methods to specialize.

Let us illustrate this approach for the `Power` class example of the previous section. The associated specialization class is defined as follows.

```
specclass Cube specializes Power {
  exp == 3;
  calculate( int b );
}
```

This specialization class specifies that the `calculate` method should be specialized with respect to the value of the `exp` field. Mentioning the `exp` field declares it as a static location, and providing a value indicates that specialization can be carried out at compile time.

If no value were supplied for the `exp` field in the specialization class, specialization would occur at run time when the `exp` field is assigned a value. In this case, the specialization class expresses the possibility to specialize `calculate` for any exponent.

Specialization classes are processed by the *Java Specialization Class Compiler*, which determines what methods should be specialized and to what values, as well as whether specialization should occur at compile time or run time. The Java Specialization Class Compiler also adds a method to the original program for switching between specialized implementations (and for generating the specialized implementations at run time if needed), and places guards that automatically invoke the method for switching implementations when a value that was used for specialization changes.

```
private void setImplementation() {
  if( this.exp == 3 )
    this.scImpl = this.getSpecImpl( "Cube" );
  else
    this.scImpl = this.getGenImpl();
}
```

An invocation of the original method is replaced by an invocation of the method currently stored in the `scImpl` field. This field can either contain the specialized method, or the original (generic) implementation.

The `Cube` specialization class specifies specialization with respect to an integer value. Specialization classes also allow invariants over object types to be expressed, by specifying the type of a field to be the name of a specialization class. These references between specialization classes allow the user to specify properties ranging across several objects. As described by Volanschi *et al.*, the dependencies between such aggregate specialization classes can be determined by the specialization class compiler, making it possible to eliminate the virtual call that is otherwise used to switch between implementations. This optimization can be essential for an efficient implementation, and is used for the specialization example of the next section.

3 Specialization Example

Object-oriented programs can be given structure and genericity by being composed from individual objects that interact through generic interfaces. As an example, we present an image filtering program. Here, multiple layers of abstraction facilitate extensibility and maintenance of a wide range of functionalities.

We use the example to illustrate specialization opportunities in object-oriented programs. In particular, the example exploits well-known abstraction mechanisms such as the strategy and abstract factory design patterns [14]. In general, the performance benefits of specialization depend on both the amount of overhead that is eliminated by specialization, and on the specialization opportunities presented by the algorithm itself.

This section first introduces image filtering, then details the structure of the example, and finally discusses the opportunities that we find for specialization.

3.1 Image Filtering

We consider image filtering based on a matrix, known as a *mask*. Filtering is performed by moving the mask across an image, computing the filtered pixel in the position of the center of the mask by performing operations on the pixels covered by the mask. Such filters can be used to obtain a variety of effects, including blurring, edge detection, and noise elimination [30]. For cache efficiency, the image is decomposed into rectangular *tiles*, and filtering is performed a tile at a time.

There are many possible representations for the data that defines an image, each representation having specific advantages. For this reason, all image data are manipulated abstractly by the filtering process, making it possible to choose the most efficient representation for an image independently of the choice of the image filter.

3.2 Structure of the Implementation

The structure of the example is shown with an object diagram in Figure 2. Only those classes that are critical to the presentation are shown. The central class `ImageFilter` manipulates data stored in a `Tile` using a pixel processing strategy defined as an `ImageOperator`, implemented by `ConvolutionOperator` and `MedianOperator`. The `ConvolutionOperator` is parameterized by a `Kernel` that defines its mask. A `Tile` stores the offsets of the tile within the image and a `DataBuffer` for holding image data. The image data processed by the operators is accessed from the `DataBuffer` through `Pixel` objects, via the data representation

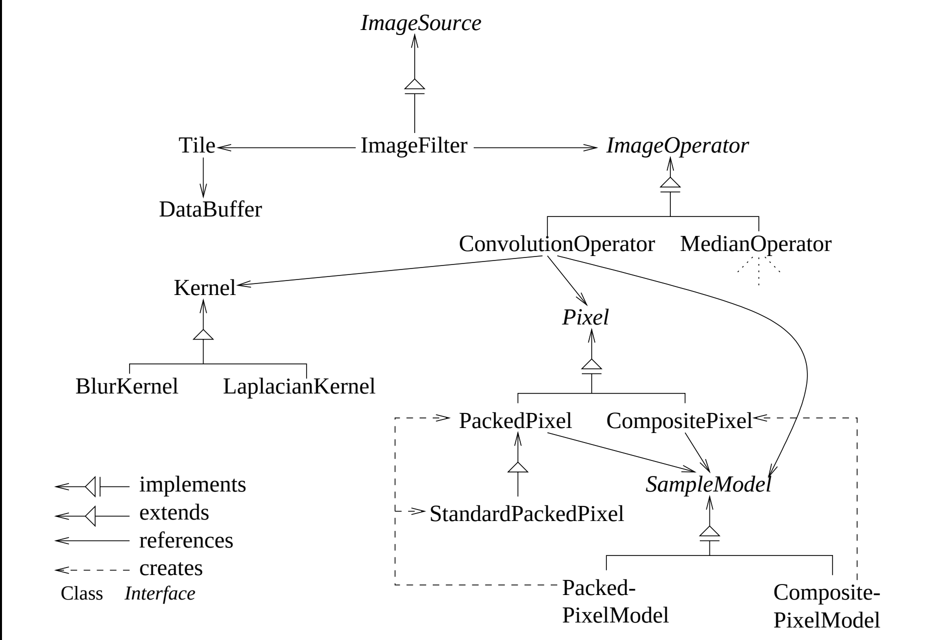

Fig. 2. Structure of the example program

strategy defined by `SampleModel`. `Pixel` objects are created using `SampleModel` as an abstract factory.

We now describe the implementation in more detail, showing the parts of the source code that will later be specialized into more efficient implementations.

The `ImageFilter` class (Figure 3) processes a tile of the image. The `getTile` method produces a filtered tile using a double `for`-loop that applies the `compute-Point` method of the filtering operator to each pixel within the source tile.

The `ConvolutionOperator` (Figure 4) and `MedianOperator` classes, which implement `ImageOperator`, each define a `computePoint` method which computes a filtered pixel.[1] In the `computePoint` method of `ConvolutionOperator`, the pixels covered by the mask[2] are traversed by a double `for`-loop, and extracted into the pixel `p`, using the `getPixel` method of the class `Tile` (Figure 5). This method uses the `SampleModel` associated with the `Pixel` to look up the data in the `DataBuffer` of the `Tile`. Finally, the pixel is scaled according to the mask, and the result is accumulated in the pixel `acc`.

[1] The convolution operator computes a linear combination of the points covered by the mask, while the median operator selects the median of the pixels covered by the mask.

[2] The specialization class compiler must guard the values stored in the mask, so the special class `ArrayOfInt` is used to represent the mask, rather than an ordinary integer array.

```
public class ImageFilter implements ImageSource {
  ImageSource src;
  ImageOperator op;
...
  public Tile getTile( int tx, int ty, int tw, int th ) {
    int b = op.getBorder();
    Tile in = src.getTile( tx-b, ty-b, tw+2*b, th+2*b );
    Tile out = in.createOutTile( tx, ty, tw, th );
    for( int y=0; y<th; y++ )
      for( int x=0; x<tw; x++ )
        out.setPixel( x, y, op.computePoint( x+b, y+b, in ) );
    return out;
  }
}
```

Fig. 3. The `ImageFilter` class. Applies an operator to each pixel of an image.

The `PackedPixelModel` class is an implementation of `SampleModel`, where multiple samples (color components, for example) are packed into a single integer value. The methods for reading a pixel (from data stored in a `Tile`) using this model are shown in Figure 5. The `PackedPixel` class is an implementation of `Pixel` that is instantiated using the `PackedPixelModel` as a factory. For a more efficient representation, the subclass `StandardPackedPixel` (Figure 6) of `PackedPixel` is instantiated when the pixel consists of three 8-bit samples. To avoid unnecessary extraction of samples from a packed pixel value, this implementation switches between a packed representation and a directly accessible representation, as indicated by the `usingSamples` field.

While the separation of the image processing algorithm into distinct classes simplifies the implementation and facilitates future extensibility, it induces a heavy performance penalty. Even when limited to a small blurring mask, our implementation performs 50 virtual calls to filter a single pixel! A hand-optimized implementation, where a dedicated filter is programmed independently for each kind of operator and data representation, would perform no virtual calls. Indeed, image processing applications are rarely structured with as much genericity as we have chosen for our application.[3] Instead, efficiency is enhanced at the price of genericity and ease of maintenance. As an alternative, program specialization can be applied to our program to enhance performance. We start by identifying the critical points that offer opportunities for optimization by specialization.

3.3 Opportunities for Specialization

The image filtering program is structured to allow flexibility in specifying the filter to be applied to the image, and the concrete representation of the pixels of

[3] For example, in the Java 2D API, it is recommended to type cast to each specific sample model, having dedicated code for each kind of representation.

```
public class ConvolutionOperator implements ImageOperator {

  int mask_width, mask_height, mask_center_x, mask_center_y;
  double mask_divisor;
  ArrayOfInt mask; // implements a simple integer array
  SampleModel model;

  ...

  Pixel acc_pixel, p_pixel; // initialized using model.getNewPixel()

  public Pixel computePoint( int x, int y, Tile inTile ) {
    int mx, my, xmax, ymax;
    int mask_element;
    ymax = mask_height - mask_center_y;
    xmax = mask_width - mask_center_x;
    Pixel acc = acc_pixel, p = p_pixel; // For memory efficiency
    acc.reset();                  // Reset pixel to neutral color

    for( my=-mask_center_y; my<ymax; my++ )
      for( mx=-mask_center_x; mx<xmax; mx++ ) {
          inTile.getPixel( x + mx, y + my, p ); // Covered by mask
          mask_element =
            mask.get(   (my+mask_center_y) * mask_width
                      + (mx+mask_center_x) );
          p.scale( mask_element ); // Scale pixel colors
          acc.add( p );            // Add pixel colors
        }

    acc.normalize( mask_divisor ); // Normalize to normal range
    return acc;
  }
}
```

Fig. 4. The ConvolutionOperator class. Combines pixels covered by the kernel.

```
public class Tile {
  int x, y, width, height; // Coordinates of Tile in image
  public DataBuffer data;
  ...
  void getPixel( int x, int y, Pixel p ) {
    p.getModel().getPixel( data, x+y*width, p );
  }
  ...
}

public class PackedPixelModel implements SampleModel {
  ...
  public void getPixel( DataBuffer d, int idx, Pixel p ) {
    ((PackedPixel)p).setValue( d.intData[idx] );
  }
  ...
}
```

Fig. 5. The `Tile` and `PackedPixelModel` classes. Storage of image data and access to image data.

the image. Nevertheless, once we begin applying a particular filter to a particular image, both the filter and the pixel representation remain invariant. Thus, the flexible program structure we have chosen presents significant opportunities for specialization.

The `ImageFilter.getTile` method (of Figure 3) can be specialized to a specific filtering operator, removing the separation between the image traversal and the action to perform for each pixel. This specialization is captured by the `ImageFilterForConvolution` specialization class of Figure 7, which specifies that a blurring operator should be used.

The `ConvolutionOperator.computePoint` method (of Figure 4) can be specialized to apply a specific convolution kernel to the image. The specialization class `BlurConvolutionOperator` of Figure 7 specifies a three-by-three kernel that takes the average of the nine surrounding pixels. The pixels manipulated by this operator are specified to be represented by a specific sample model.

An operator manipulates image data through methods in `Tile` and the concrete `Pixel` implementation. These methods make use of a concrete `SampleModel` class to manipulate the raw data stored in the `DataBuffer` of the `Tile`. Specializing the operator to a concrete `SampleModel` class exposes the concrete type of pixels, and allows the data in the `DataBuffer` to be directly manipulated. The specialization class `RGB8bitPixelModel` of Figure 7 captures the common case of a `PackedPixelModel` with three 8-bit samples used to represent a pixel (i.e., red, green, and blue). This specialization class implies that the `PackedPixelModel` always instantiates pixels of type `StandardPackedPixel` (by the implementation of the concrete `Pixel` factory, not shown).

```
public class StandardPackedPixel extends PackedPixel {
  int value;
  boolean usingSamples;          // When false, 'value' is used
  int sample1, sample2, sample3; // Three 8-bit samples
  public void setValue( int v ) {
    value = v; usingSamples = false;
  }
  void initializeSamples() {
    int pixel = value;
    sample1  = (pixel & 0xff0000) >> 16;
    sample2  = (pixel &   0xff00) >>  8;
    sample3  = (pixel &     0xff);
    usingSamples = true;
  }
  public void scale( int s ) {
    if( !usingSamples ) initializeSamples();
    sample1 *= s; sample2 *= s; sample3 *= s;
  }
... // Other methods implementing the Pixel interface
}
```

Fig. 6. The `StandardPackedPixel` class. Representation dedicated to 8-bit pixels with three samples.

```
specclass ImageFilterForConvolution specializes ImageFilter {
  BlurConvolutionOperator op;
  void getTile( int tx, int ty, int tw, int th );
}
specclass BlurConvolutionOperator specializes ConvolutionOperator {
  mask_width == 3;
  mask_height == 3;
  mask_center_x == 1;
  mask_center_y == 1;
  mask_divisor == 9.0;
  mask == {1,1,1,1,1,1,1,1,1};
  RGB8bitPixelModel model;
  void computePoint( int x, int y, Tile inTile );
}
specclass RGB8bitPixelModel specializes PackedPixelModel {
  numberOfSamples == 3;
  bitsPerSample == 8;
}
```

Fig. 7. Declaration of specialization opportunities.

The specialization classes of Figure 7 are linked together so that when the `getTile` method is called, the concrete `ImageFilter` object is in a specialized state only if the aggregate objects described by the aggregate specialization classes are also in a specialized state. Thus, a virtual call to make the choice of implementation is only necessary when calling the `getTile` method.

We next describe how we realize these specialization opportunities.

4 Specializing Java Programs

Traditionally, program specialization optimizes a program based on input values provided by the user, and on constants explicit in the program. In the case of Java, specialization can exploit additional static information, such as the type of each object, and can simplify operations implicit to the virtual machine.

Throughout this section, for readability, we illustrate the result of specialization using Java code, rather than the equivalent C code actually produced automatically by our implementation.

4.1 Specializing Data Encapsulation

In an object-oriented language such as Java, values are systematically encapsulated. As a consequence, accessing a value is implemented in terms of a sequence of pointer dereferences whose cost depends on the embedding depth of the object structure. Specialization replaces a reference to a field containing a static value by the value itself. This transformation eliminates memory references and improves performance.

Example. The specialization class `BlurConvolutionOperator` (Figure 7) indicates that `computePoint` (Figure 4) should be specialized with respect to the dimensions and weights of the mask. Specialization propagates these constants, eliminating references to these fields, and triggering other optimizations.

Because the dimensions of the mask control the double `for`-loop of `computePoint`, it can be unrolled during specialization. The resulting code consists of nine (i.e., `mask_width` $*$ `mask_height`) blocks of code, each specialized according to the current loop indices. These known indices allow the specializer to evaluate the references to the weights of the mask, thus replacing these array references by constants. The result of this specialization is illustrated in Figure 8.

4.2 Specializing Object Types

In an object-oriented language such as Java, control flow depends on object types as well as program values. The choice of which method is invoked by a Java method call depends on the type of the receiver object. When the definition of the invoked method cannot be determined at compile time, the method call is said to be *virtual* and is typically implemented using a table and a pointer

```
public Pixel computePoint( int x, int y, Tile inTile )
{
  int mask_element;
  Pixel acc = acc_pixel, p = p_pixel;
  acc.reset();
  {
    inTile.getPixel( x + (-1), y + (-1), p );
    p.scale( 1 );
    acc.add( p );
  }
  // Repeats nine times, with different arguments to getPixel
...
  acc.normalize( 9.0 );
  return acc;
}
```

Fig. 8. The `computePoint` method specialized for data values.

dereference. If the type of the receiver object can be determined based on extra information available during specialization, then specialization replaces a virtual method call by an ordinary procedure call. Such a procedure call can be inlined, leading to further optimizations.

Example. The `getTile` method of the `ImageFilter` class (Figure 3) contains a virtual call `op.computePoint` in a doubly nested loop. The `ImageFilterForConvolution` specialization class of Figure 7 specifies that `op` has type `BlurConvolutionOperator`. Specializing `ImageFilter` to the type of `op` replaces the virtual call `op.computePoint` by a direct call to the `computePoint` method of the specialized `ConvolutionOperator` class, thus enabling subsequent inlining. (To save space, this optimization is not illustrated in the figures.)

The use of the sample model in the `computePoint` method provides a more dramatic example. The specialization class `BlurConvolutionOperator` specifies that pixels are represented according to the specialization class `RGB8bitPixelModel`. This specialization class implies that the sample model is a `PackedPixelModel` class that creates pixels using the optimized `StandardPackedPixel` representation. Thus, specialization replaces virtual calls performing pixel operations with direct calls to the methods of a `StandardPackedPixel` object. Furthermore, the implementation of `StandardPackedPixel` is optimized to choose between two representations, depending on how the pixel is used. Because specialization can determine the control flow through these methods, it can eliminate the overhead (specifically, the `usingSamples` field) associated with maintaining two possible representations. The transformed method, which can be viewed as a more specialized version of Figure 8, is illustrated in Figure 9. The C code actually produced by specialization can be compiled using an optimizing C compiler, which (for example) eliminates the multiplications by 1.

4.3 Specializing the Virtual Machine

Java requires run-time checking to be performed by the virtual machine to ensure some safety properties. In particular, casts and array references are systematically checked. An object can only be cast to a type that is a supertype of the actual object type. An array reference must access an element within the array bounds. In general, these tests must be carried out at run time. When the specializer can determine either the type of the object, or the size of the array and the index of the accessed element, the tests can be eliminated during specialization. Unfortunately, by definition of Java bytecode, these optimizations cannot be expressed at the bytecode level. Since we directly execute the specialized C code using the Harissa environment, our approach does not suffer from this problem. Concretely, the specialized program is as efficient as an equivalent C program, but with the safety of the original Java program, at the price of Java portability.

Example. After having specialized the `computePoint` method with respect to both values and types, a number of type casts remain for the pixels `acc` and `p`. Since the types of both pixels are static, these type casts can be checked and eliminated during specialization. This is not shown in Figure 9, since we cannot show this at the Java level.

4.4 Result of Specialization

The original image filtering program is essentially constructed from a collection of generic, interacting objects. Program specialization automatically adapts each object to its context, obtaining an implementation very similar to a hand-optimized version. In the case of our example, specialization produces an optimized implementation adapted to a specific filtering strategy.

Experiments with Harissa code. The performance of the specialized code is significantly better than that of the generic, unspecialized code. We have tested the automatically specialized code using the Harissa environment. Experiments were conducted both on a Sun workstation (300MHz Ultra SPARC) and on a Dell PC (200MHz Pentium Pro). The results, including both the execution time and the speedup of the specialized code, are shown in Figure 10.

We conducted experiments on a three-by-three blurring convolution filter, a seven-by-seven blurring convolution filter, and a median filter. The speedups are between 1.55 and 4.00, the highest speedup being obtained for the large convolution filter. The relatively minor speedup for the median filter is derived primarily from structural simplifications, the running time being dominated by the median computation. For the convolution filters, the code is simplified to such a degree that memory access becomes the major bottleneck.

```
public Pixel computePoint( int x, int y, Tile inTile )
{
  Pixel acc = acc_pixel, p = p_pixel;
  ((StandardPackedPixel)acc).value = 0;
  ((StandardPackedPixel)acc).usingSamples = false;

  {
    // inTile.getPixel( x + (-1), y + (-1), p )
    ((StandardPackedPixel)p).value =
      inTile.data.intData[(x-1)+(y-1)*intTile.width];

    // p.scale( 1 )
    {
      int pixel = ((StandardPackedPixel)p).value;
      ((StandardPackedPixel)p).sample1  = (pixel & 0xff0000) >> 16;
      ((StandardPackedPixel)p).sample2  = (pixel &   0xff00) >>  8;
      ((StandardPackedPixel)p).sample3  = (pixel &     0xff);
      ((StandardPackedPixel)p).usingSamples = true;
    }
    ((StandardPackedPixel)p).sample1 *= 1;
    ((StandardPackedPixel)p).sample2 *= 1;
    ((StandardPackedPixel)p).sample3 *= 1;

    // The specialized code for acc.add( p )
    ...
  }

  // This block repeats nine times, with different array indices
  ...
  // Afterwards, the specialized code for acc.normalize( 9.0 );
  ...

  return acc;
}
```

Fig. 9. The `computePoint` method specialized for types.

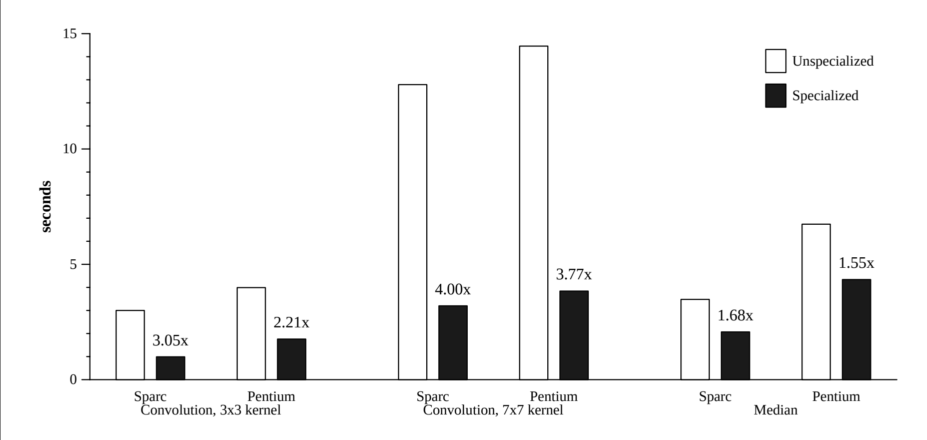

Fig. 10. Automatically specialized programs running under Harissa.

Experiments with specialized Java source code. To estimate the gains that can be expected from a program specializer that outputs Java bytecode programs, we have manually translated the specialized C code into Java. The benefits due to virtual machine specific optimizations have been retained where possible, mimicking an optimized translation process rather than a direct, naive one. The tests were executed using JDK 1.2b4 (reference version) on the same machines as were used in the previous experiment. Figure 11 shows the result of our experiments (note that the scale has changed).

The speedups are between 1.44 times and 5.13 times, resembling the speedups of the automatically specialized programs. The speedups due to specialization are a bit higher than those that were observed using Harissa. Since the JIT executes programs more slowly than the Harissa environment, memory access is not as dominating a factor as with the Harissa tests, making the benefits of the code simplifications performed by the specializer more apparent.

5 Description of the Prototype

Rather than writing a program specializer from scratch, we have chosen to construct a prototype from existing tools, minimizing the amount of work needed to perform realistic initial experiments. The prototype is implemented as a staged process. First, the Java program and the specialization classes are processed to prepare for specialization and produce the corresponding run-time environment. Then, the Java code is translated into C. Finally, the C code is specialized using the Tempo specializer.

We have seen that taking the approach of translating Java programs into C allows aspects of the semantics of Java to be made explicit, which exposes specialization opportunities. Examples include virtual calls, casts, and array references (see Section 4). Thus, following our approach, the kinds of specialization

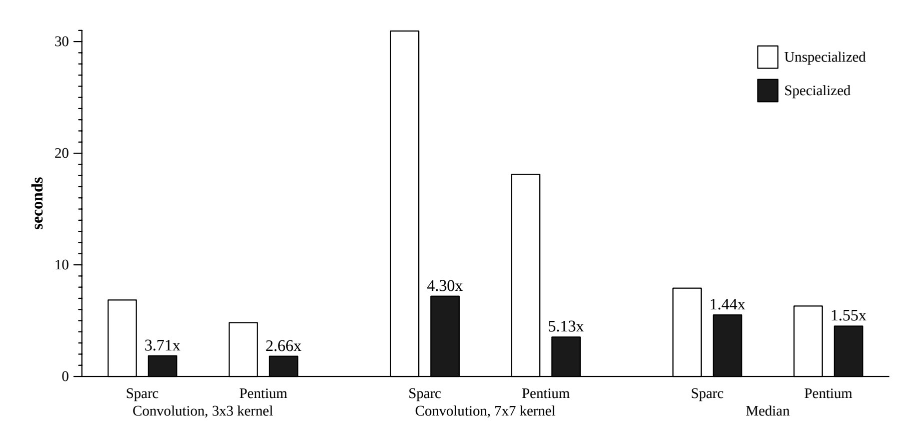

Fig. 11. Prospective benchmarks. Manual back-translation from C to Java.

that can be expressed using this approach are not limited by the syntax and semantics of Java, but by the expressiveness of C. However, there is a possible loss of precision when analyses are performed in C as opposed to Java. For example, alias analysis in Java need not consider incorrectly-typed aliases. Should this loss of precision pose a problem, we could perform the alias analysis before translation to C, and generate annotated C code specifically for Tempo.

5.1 Structure of the Prototype

The overall structure of the prototype is presented in Figure 12. The input to the specialization process is a Java program and a set of specialization classes. The Java Specialization Class Compiler (`JSCC`) [32] prepares the program for specialization. For each class to be specialized, it generates a "dispatcher" to select the appropriate (specialized or generic) implementation to invoke with respect to the execution context; this is called an enclosing object (X-enclosing.java). Additionally, `JSCC` produces stub methods to (native) specialized implementations of this object (X-specializable.java) that will be generated by Tempo in a later phase.

Next, the extended Java program is compiled into bytecode using `javac` and translated into C via the Harissa compiler.

Tempo takes the C-translated program together with the specialization parameters and generates the specialized versions. The C-translated program and the specialized code can either be translated back into Java, at the expense of losing some optimizations, or be run directly using the Harissa environment.[4] The latter option is the only one currently implemented. Note that Java programs can be specialized at both compile time and at run time.

[4] Specialization simplifies the program without adding new code, so the specialized program is still compatible with the Harissa environment.

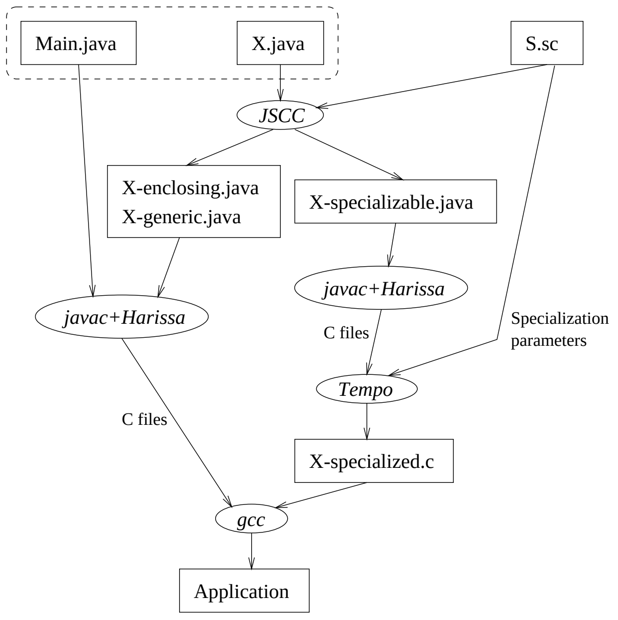

Fig. 12. Structure of the prototype, specialization of class `X`

5.2 Harissa

Harissa is an off-line compilation system that supports dynamic loading of bytecode. It compiles Java bytecode into C code that can be further compiled by standard C compilers. Harissa includes aggressive optimizations, such as method inlining, driven by a Class Hierarchy Analysis and an intra-procedural static class analysis [11]. These optimizations are complemented by the C compiler. Harissa's run-time system integrates a Java bytecode interpreter to execute dynamically loaded bytecode.

Harissa is designed to compile a Java program into C code while preserving as much of the original structure as possible. Preserving the structure of the program makes Harissa suitable for generating programs that are to be processed by a back-end phase such as Tempo. An object is represented using an ordinary C structure, and a virtual call is expressed as an indirect C function call through a pointer stored in a structure field, allowing it to be precisely treated by Tempo, and replaced by a direct call when the pointer is static. Harissa has been extended to generate extra information useful to Tempo. In particular, Harissa generates a context for the specialization containing information about those parts of the program that are not being specialized.

Currently, Harissa implements exceptions using the C `longjmp` function, which is ignored by Tempo. Specializing in the presence of exceptions can thus produce incorrect results. In the future, we plan to extend Tempo with explicit treatment of exceptions, extending the C input language to include a Java-like exception mechanism, and then to have Harissa generate C code with such exceptions.

5.3 Tempo

Tempo is an off-line program specializer for the C language [8], that supports both compile-time and run-time specialization. The analyses of Tempo are sufficient to effectively specialize a wide range of C programs. We found it necessary, however, to improve the precision of some of the analyses to better specialize Java programs.

The C code generated from a Java program is very different from human-written C programs. In particular, because objects are represented as C structures, structures are used more heavily in Harissa generated code than in ordinary C programs. Also, memory management in C is different from memory management in Java, where all objects are dynamically allocated. Because of this situation, we extended Tempo with a more precise treatment of structures.

Upward type casts are implicit in Java, and type casts between objects are common. In contrast, they rarely occur in C and are often considered as a bad programming style. To address this need, Tempo has been extended to handle type casts between structures. This capability is geared towards the programs generated by Harissa, in which the layout of structures is guaranteed to be compatible.

6 Related Work

To our knowledge, there have been only preliminary investigations of specialization of object-oriented programs. We apply the approach of translation to an intermediate language for which a specializer exists; this approach has been investigated earlier to implement program specialization for a simple imperative language. Also, optimizing compilers have addressed specialization of both data representation and control flow.

Specialization of object-oriented languages. Marquard and Steensgaard developed a partial evaluator for a small object-oriented language based on Emerald [22]. They focused primarily on implementation issues such as ensuring termination and the representation of unknown values during the specialization process. In contrast, we have investigated the applicability of program specialization to the object-oriented paradigm.

Khoo and Sundaresh investigate partial evaluation as a means for eliminating virtual calls in simple object-oriented language [17]. Their work focuses on formalizing the analysis and transformations realized by performing program specialization on programs with virtual calls.

The C++ language incorporates templates that allow a single class to be statically compiled to different types, in effect performing specialization. Templates can also be used to perform some computations at compile time, as demonstrated by Veldhuizen [31], albeit not on object types, and with a significant compilation overhead. Special syntax must be used to write programs so that they can be optimized using templates, and having both specialized and generic variants of a method requires writing two different implementations.

Specialization via translation. Moura has also investigated the approach of using translation to extend the applicability of an existing program specializer to new program constructs [23]. She designed an approach to specializing imperative programs by translation into a functional subset of Scheme, followed by specialization using an existing specializer for Scheme, named Schism [7].

Optimization of data representation. Object inlining is a technique for storing temporary objects on the stack [12]. An object that is used only locally in a method can be stack-allocated rather than heap-allocated, thus improving the performance of the program.

The control flow simplification performed by specialization enables further optimizations on data structures. Object inlining is an instance of such optimizations that we will consider in the future.

Optimization of control flow. The Vortex compiler [10] implements optimizations similar to those performed by program specialization. Vortex is based on static, global analyses, complemented by an automated profiling system. Profiling information guides aggressive optimizations. These optimizations eliminate virtual calls and specialize methods according to the types of their arguments. In fact, the optimizations offered by Vortex depend on the accuracy of its analyses and profiling system. As a result, the level of optimization is difficult to predict. By contrast, specialization is parameterized by information provided by the programmer (or component user).

Furthermore, Vortex's optimizations only propagate and exploit type information to remove virtual calls. Specialization goes beyond types, also propagating values. As a result, more optimizations can be performed with these values (*e.g.,* loop unrolling and array bounds checking).

With Vortex's approach, all compilation is performed before the program is run. Rather than considering compilation, execution, and profiling as separate phases, the execution environment can include all these tasks, and perform compilation as a continuous process. This approach allows aggressive optimizations similar to those employed by program specialization, since information in the form of usage patterns is available at run time [16]. However, the analyses that can be performed are inherently limited by the amount of available time and space. Also, checks must be retained to verify invariants that could have otherwise been detected by a specializer.

7 Future Work

We have provided an outline of how automatic program specialization of an object-oriented language such as Java can be achieved. Program specialization of Java as presented in this paper has many applications and raises many issues. This section outlines possible extensions to our work.

Software components. A trend in software engineering is the development of systems from software components. This emerging software architecture is illustrated by Java Beans and its rapidly growing selection of components. This trend stresses the need for genericity to address a class of solutions by a specific software component. To overcome the expected performance penalty, specialization will likely become a critical tool.

In this context, our next step aims at developing a methodology for designing and developing highly-generic components, that, once integrated, specialize in a predictable way into efficient implementations.

Java as target language. For portability, it can be beneficial to produce specialized code in Java. One option is to translate the specialized C program back into Java. Nevertheless, such an approach would eliminate some optimizations, mainly those related to the virtual machine (see Section 4.3).

Translation back into Java is possible if the C specialized code has enough information and structure to enable Java constructions to be recovered. Given our specialization process, this depends on the translations performed by the Harissa compiler and the transformations done by Tempo. Motivated by the encouraging benchmark results in Section 4.4, future work includes the development of such a back translator.

Run-time specialization. Although the current implementation of the specialization class compiler does not support run-time specialization, Tempo does support specialization at both compile-time and run-time. For run-time specialization, executable code is assembled directly from binary templates generated by a C compiler. This approach can be used directly in the Harissa environment. Specializing at run time has the obvious advantage of exploiting values that are only available when running the program.

8 Conclusion

Component-based software technology is a growing trend in software development. It makes genericity a central issue. In this paper, we have demonstrated that specialization is a key tool to overcome the performance penalty incurred by genericity.

Specialization exploits global information available when software components are integrated into an application. Performing transformations such as virtual-call elimination, inlining, constant propagation and constant folding turns a modular component-based application into a monolithic optimized program.

We have developed a specializer for Java based on existing tools, namely, a Java-to-C compiler (Harissa) and a C specializer (Tempo). We have used it to specialize an image-filtering application. This application is structured in a modular way to support a variety of data representations and image treatments. Specialization has been shown to eliminate the overhead incurred by this structuring strategy. In practice, the specialized program is up to four times as fast as the original one.

Several lessons can be drawn from this work.

– Specialization of modular Java programs can drastically improve performance. Besides the usual optimizations offered by imperative specializers, we have found that object-oriented programs offer other opportunities for improvement, opportunities traditionally studied as advanced optimizing compilation techniques.
– Re-using existing tools to develop our Java specializer has proved successful in terms of development time. This result has been obtained largely without compromising the quality of the specialized program.
– Because our approach involves specializing C programs, it enables a class of optimizations that is out of reach of ordinary Java source-to-bytecode compilers.

We believe that a specializer is a key tool for object-oriented software development environments. In particular, in the context of Java Beans, a specializer should allow modular applications to be mapped into efficient implementations.

Availability. Tempo and the Java Specialization Class Compiler are freely available. Harissa is available under the GNU Public license (with full source code). More information can be found at the Compose home page:
`http://www.irisa.fr/compose`

Acknowledgements. Thanks to the anonymous referees for their valuable comments, to Renaud Marlet for helping to streamline the paper, and to Aino Rasmussen for her encouraging comments.

References

1. L.O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, Computer Science Department, University of Copenhagen, May 1994. DIKU Technical Report 94/19.
2. J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI'96 [27], pages 149–159.
3. R. Baier, R. Glück, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.

4. A.A. Berlin. Partial evaluation applied to numerical computation. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, Nice, France, 1990. ACM Press.
5. B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.
6. W.H. Cheung and A. Loong. Exploring issues of operating systems structuring: from microkernel to extensible systems. *ACM Operating Systems Reviews*, 29(4):4–16, October 1995.
7. C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.
8. C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in Lecture Notes in Computer Science, pages 54–72, February 1996.
9. C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23^{rd} *Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996. ACM Press.
10. J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. In *OOPSLA' 96 Conference*, pages 93–100, San Jose (CA), October 1996.
11. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP '95*, Aarhus, Denmark, August 1995. Springer-Verlag.
12. J. Dolby and A. A. Chien. An evaluation of automatic object inline allocation techniques. In *Proceedings OOPSLA '98 Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices. ACM, 1998.
13. D.R. Engler and M.F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 26–30, Stanford University, CA, August 1996. ACM Press.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
15. B. Guenter, T.B. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.
16. U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 326–336, New York, NY, USA, June 1994. ACM Press.
17. Siau Cheng Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 211–222, New Haven, CT, USA, September 1991. ACM SIGPLAN Notices, 26(9).
18. P. Kleinrubatscher, A. Kriegshaber, R. Zöchling, and R. Glück. Fortran program specialization. In U. Meyer and G. Snelting, editors, *Workshop Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, pages 45–54. Justus-Liebig-Universität, Giessen, Germany, 1994. Report No. 9402.
19. P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI'96 [27], pages 137–148.

20. B.N. Locanthi. Fast bitblt() with asm() and cpp. In *European UNIX Systems User Group Conference Proceedings*, pages 243–259, AT&T Bell Laboratories, Murray Hill, September 1987. EUUG.
21. R. Marlet, S. Thibault, and C. Consel. Mapping software architectures to efficient implementations via partial evaluation. In *Conference on Automated Software Engineering*, pages 183–192, Lake Tahoe, Nevada, November 1997. IEEE Computer Society.
22. M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, University of Copenhagen, Department of Computer Science, Universitetsparken 1, 2100 Copenhagen O., Denmark, April 1992.
23. B. Moura. *Bridging the Gap between Functional and Imperative Languages.* PhD thesis, University of Rennes I, April 1997.
24. G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 240–249, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.
25. G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Portland (Oregon), USA, June 1997. Usenix.
26. G. Muller and U. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, March 1999.
27. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996. ACM SIGPLAN Notices, 31(5).
28. C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 314–324, Copper Mountain Resort, CO, USA, December 1995. ACM Operating Systems Reviews, 29(5), ACM Press.
29. C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.
30. J.C. Russ. *The Image Processing Handbook.* CRC Press, Inc., second edition, 1995.
31. T. L. Veldhuizen. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18, San Antonio, TX, USA, January 1999. ACM Press.
32. E.N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, USA, October 1997. ACM Press.