Towards Automatic Specialization of Java Programs

Ulrik Pagh Schultz, Julia Lawall, Charles Consel, Gilles Muller

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Towards Automatic Specialization of Java Programs

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

N ° 3579

Décembre 1998

THÈME 2

ISSN 0249-6399

Towards Automatic Specialization of Java Programs

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

Thème 2 — Génie logiciel
et calcul symbolique
Projet COMPOSE

Rapport de recherche n3579 — Décembre 1998 — 24 pages

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 domain of imperative languages, but so far it 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-times 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.

Key-words: Java, program specialization, object-oriented programming, software components

(Résumé : tsvp)

This work is supported in part by BULL.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Vers la spécialisation automatique de programmes Java

Résumé : La spécialisation automatique de programme permet de générer automatiquement des logiciels optimisés à partir de composants génériques. En cela, elle permet de reconcilier les objectifs opposés de généricité et d'efficacité. L'utilité de cette technique a démontrée dans le domaine des langages impératifs, mais celle-ci n'a pas encore été étudiée dans le domaine des langages orientés objets.

Dans cet article, nous présentons une expérimentation dans la spécialisation de programmes Java. Nous montrons qu'il est possible de construire un spécialiseur pour Java à partir d'un spécialiseur pour programmes C et d'un compilateur Java vers C. Dans notre approche, la spécialisation est contrôlée via une approche déclarative qui permet d'abstraire le processus d'optimisation et de cacher les détails de mise en oeuvre. Sur la base de nos études, nous avons identifié les optimisations de programmes objets qui peuvent être réalisées par spécialisation de programmes. En particulier, nous avons obtenu un facteur d'accélération de quatre sur une application de filtrage d'image.

Ces résultats nous permettent d'affirmer que la spécialisation de programmes est une technique prometteuse pour le domaine des composants logiciels; il est désormais possible d'optimiser un composant object générique à un contexte d'utilisation donné.

Mots-clé : Java, spécialisation de programmes, programmation orientée objet, composants logiciels

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 [18]. This conflict is being increasingly addressed, with success, using forms of *program specialization* [19]. 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, 26, 27]. 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, 2, 3, 7, 8, 16, 17]. Applications of program specialization are emerging in a number of fields, including scientific code [3, 4, 8], systems software [13, 22], and computer graphics [14], with very promising results. However, automatic specialization of object-oriented programs remains uninvestigated. Given the existing base of experience with imperative languages, we aim at showing that a program specializer can be used 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 a slightly modified version of the Harissa optimizing Java-to-C compiler [23, 24].

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.* [29] to specify the specialization strategies in a high-level manner.
- 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 problem 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 demonstrates specialization. Then, Section 3 introduces our example program. This program is specialized in Section 4, which defines the actual process of performing specialization of 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 like Fortran [3] and C [1, 8] have been developed. Real-sized applications of program specializers to various areas like systems software [22] and scientific computing [3] have clearly demonstrated that automatic program specialization is a realistic and 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, we hope to enable further optimization. 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). Partial evaluation divides values into two categories: *static* values are those that are available during the specialization process, *dynamic* values are those that are not available until after specialization. Computations that only depend on static values are themselves said to be static, and are performed by the

specializer. The dynamic computations, that may depend on dynamic values, are reconstructed and constitute the specialized program.

Partial evaluation aggressively propagates input values specified by the user, as well as constants explicit in the program, to perform constant folding. Examples of transformations include loop unrolling and some forms of strength reduction. Partial evaluation is different 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.

The set of values to specialize over may become gradually available during compilation and execution. As a result, programs may need to be specialized both at compile time and at run time. In fact, the partial evaluator Tempo offers both strategies in a uniform approach. Its run-time specialization capability has shown to produce efficient programs and to be amortized in a few runs of the specialized program [9]. Concretely, these features widen the opportunities to eliminate genericity in programs.

The process of specializing parts of a large program, and then reorganizing the program to use the specialized code is complex. In the software development process, a programmer would rather specify the various scenarios for specializing a program instead of directly invoking the specialization engine. Thus we need a language for declaring specialization opportunities.

2.2 A Declarative Approach to Program Specialization

Volanschi *et al.* [29] 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*. More specifically, a specialization class defines the conditions under which a specialization should occur and what methods to specialize.

Let us illustrate this approach 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 `3`. The corresponding specialization class, named `Cube`, 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. Providing a value for `exp` declares it as a static value, and also specifies that specialization can be carried out at compile time. By unrolling the loop in the `calculate` method and computing the values of the exponent, the specialization process generates the following method.

```
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;
  }
}
```

Figure 1: A Java class for the power function.

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

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 `Power` for any exponent.

Specialization classes are processed by the *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 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 methods for switching implementations when a value that was used for specialization changes.

```
private void setImplementation() {
  if( this.exp == 3 )
    this.scImpl = this.getSpecImpl( "Cubed" );
  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.

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 of such a program, we present an image filtering program. Here, multiple layers of abstraction facilitate extensibility and maintenance of a wide range of functionalities. An image is viewed as an abstract source of pixels. Sources are filtered by abstract operators that are applied to each pixel. Pixel operations are decoupled from their low-level representation by an abstract interface.

This section first introduces image filtering, then details the structure of the example, and finally discuss 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 [28]. For efficiency, the image is decomposed into rectangular *tiles*, and filtering is performed a tile at a time.

Conceptually, filtering is performed by applying an operator to an image source, yielding a new image source. All operators filter pixels in a similar fashion, and are thus defined in terms of a standard interface. The behaviour of each operator is defined by its parameters, such as the mask that is to be applied to the image source.

There are a variety of representations for the data that defines an image, each representation having specific advantages and disadvantages. For this reason, all image data and specific pixels extracted from an image are manipulated as abstract data by the filtering process, making it possible to choose the most efficient data 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`. The data is stored in a `DataBuffer` and is accessed via the data representation strategy defined by `SampleModel`. The operator implementation `ConvolutionOperator` is parameterized by a `Kernel` that defines its mask. This operator manipulates `Pixel` objects that are instantiated using `SampleModel` as an abstract factory.

The `ImageFilter` class processes the image in tiles, and provides a method `getTile` to obtain a tile. This method is shown in Figure 3. Here, a double `for`-loop defines the pixels in the output (filtered) tile, by invoking the `computePoint` method of the operator that is being applied to the source.

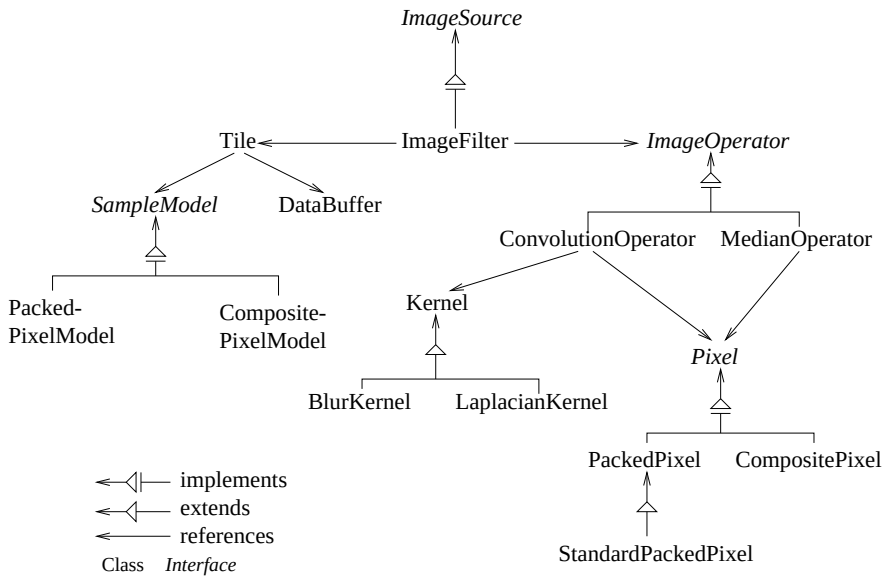

Figure 2: Structure of the example program

```
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;
  }
}
```

Figure 3: The `ImageFilter` class. Traverses image while applying operator.

```
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 by 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;
  }
}
```

Figure 4: The `ConvolutionOperator` class. Combines pixels covered by kernel.

The `ConvolutionOperator` and `MedianOperator` classes are `ImageOperator` implementations, each providing a method `computePoint` to obtain a computed pixel.[1] The `computePoint` method is shown for the `ConvolutionOperator` class, in Figure 4. The pixels covered by the mask are traversed by a double `for`-loop, and extracted into the pixel `p`, using the `getPixel` method of the class `Tile`. This method uses the `SampleModel` associated with the `Tile` 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`.

The `PackedPixelModel` class is an implementation of `SampleModel`, where multiple samples (color components, for example) are packed into a single integer value, each integer value being an element in a one-dimensional array stored in the `DataBuffer`. The `PackedPixel`

[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.

```
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 for implementing the Pixel interface
}
```

Figure 5: The `StandardPackedPixel` class. Representation dedicated to 8-bit pixels with three samples.

class is an implementation of `Pixel` that is instantiated using the `PackedPixelModel` as a factory. For a more efficient representation, the subclass `StandardPackedPixel` of `PackedPixel` is instantiated when the pixel consists of three 8-bit samples. The `StandardPackedPixel` class is shown in Figure 5.

While the separation of the image processing algorithm into distinct classes offers many advantages, it induces a heavy performance penalty. Even when limited to a small blurring mask, our implementation will perform 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.[2] 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.

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

```
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};
  SpecPackedPixelModel model;
  void computePoint();
}
specclass RGB8bitPixelModel specializes PackedPixelModel {
  numberOfSamples == 3;
  bitsPerSample == 8;
}
```

Figure 6: Declaration of specialization opportunities.

3.3 Opportunities for Specialization

The image filtering program is structured so as to allow flexibility in specifying the filter to be applied to the image, and the concrete representation of the pixels of 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, obtaining behaviour specific to this operator. This specialization is captured by the `ImageFilterForConvolution` specialization class of Figure 6, which specifies that a specific blurring operator should be used.

The `ConvolutionOperator.computePoint` method (of Figure 4) can be specialized to apply a specific convolution kernel to the image, obtaining a dedicated operator. The specialization class `BlurConvolutionOperator` of Figure 6 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`. These methods make use of a concrete `SampleModel` class to manipulate the raw data stored in the `DataBuffer` of the `Tile`. The operator can be specialized to a concrete `SampleModel` class, allowing data in the `DataBuffer` to be directly manipulated. The `SampleModel` class also serves as an abstract factory to produce the concrete implementation of pixels. Specializing for a concrete `Sample-`

`Model` thus exposes the concrete type of the pixels, enabling further optimizations. The specialization class `RGB8bitPixelModel` of Figure 6 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 will instantiate pixels of type `StandardPackedPixel`.

We next describe how we realize these specialization opportunities.

4 Specializing Java Programs

Traditionally, partial evaluation aggressively propagates input values specified by the user, as well as constants explicit in the program, to perform constant folding. In the case of Java, there is additional information in the source program that may be used at specialization time. This information includes the type of each object and operations implicit to the virtual machine.

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 references to a field containing a static value by the value itself. This optimization eliminates memory references and can significantly improve performance. The static value can, of course, also trigger further optimization.

Example

The `computePoint` method of the `ConvolutionOperator` class, shown in Figure 4, benefits from specialization of data encapsulation. The specialization class `BlurConvolutionOperator` displayed in Figure 6, indicates that `computePoint` should be specialized with respect to the dimensions and weights of the mask.

Because the dimensions of mask control the double `for`-loop of `computePoint`, it can be unrolled during specialization. The resulting code consists of nine (`mask_width * mask_height`) blocks of code, each of which can be specialized according to the current loop indices. These known indices allow the specializer to evaluate the references to the elements of the mask, thus replacing these array references by constants. The result of this specialization is illustrated in Figure 7.

4.2 Specializing Object Types

In an object-oriented language such as Java, control flow depends on program values as well as object types. 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

```
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 );
    mask_element = 1;
    p.scale( mask_element );
    acc.add( p );
  }
  // Repeats nine times, with different arguments to getPixel
...
  acc.normalize( 9.0 );
  return acc;
}
```

Figure 7: The `computePoint` method specialized for data values.

at compile time, the method call is said to be *virtual* and is typically implemented using a table and a pointer dereference. Thus, method invocation includes an implicit type check. 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

As a simple example of specialization with respect to types, consider the `getTile` method of the `ImageFilter` class, shown in Figure 3. Because methods of the object `op` are extensively invoked, `getTile` is an ideal candidate for specialization with respect to the type of `op`. The `ImageFilterForConvolution` specialization class of Figure 6 specifies that `op` has type `BlurConvolutionOperator`, which is a specialized variant of `ConvolutionOperator`. Specializing `ImageFilter` to the type of `op` replaces the virtual call to `op.computePoint` by a direct call to `BlurConvolutionOperator` class.

A more advanced example can be found in the `computePoint` method, where the representation of pixels and the access to the `Tile` are controlled by the `SampleModel`. Not only can the `computePoint` method be specialized with respect to the values of certain fields, as shown in the previous section, but it can also be specialized with respect to the type of the `model` field that holds the `SampleModel`, as shown in the specialization class `BlurConvolutionOperator` of Figure 6. Specifying the exact representation of the data, as is done in the specialization class `RGB8bitPixelModel`, enables further transformations by the specializer, since an optimized representation of pixels can be used in this case.

```
public Pixel computePoint( int x, int y, Tile inTile )
{
  Pixel acc = acc_pixel, p = p_pixel;
  ((StandardPackedPixel)acc).value = 0;
  ((StandardPackedPixel)acc).usingSamples = false;
  {
    ((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 );
  }
  // Repeats nine times, with different array indices
...
  // The specialized code for acc.normalize( 9.0 );
  return acc;
}
```

Figure 8: The `computePoint` method specialized for types.

Operations on the tile are defined by virtual calls to the `SampleModel` defining the layout. By specializing with respect to the actual pixel representation, these operations can now be expressed directly as operations on the arrays that hold the actual data. The transformed method, which can be viewed as a more specialized version of Figure 7, is illustrated in Figure 8.

4.3 Specializing the Virtual Machine

Java requires some 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 may 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

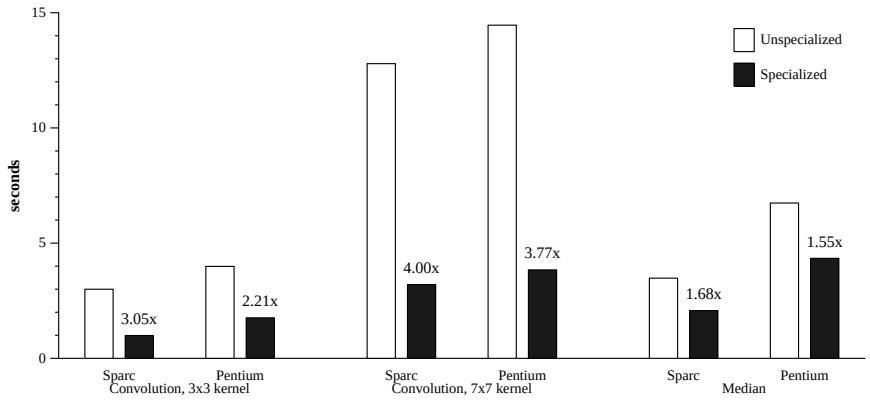

Figure 9: Automatically specialized programs running under Harissa.

the object, or the size of the array and the index of the accessed element, the tests can be eliminated during specialization. Unfortunately, by the definition of Java bytecode, these optimizations cannot be expressed at the bytecode level. However, our approach does not suffer from this problem, if we directly execute the specialized C code using the Harissa environment. 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 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.

4.4 Result of Specialization

A well-structured object-oriented program is typically composed of a set of generic components. Specializing such a program adapts each generic component to a given context. Furthermore, specialization produces a monolithic program where independent components have been merged.

Example

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 practice, specialization produces an optimized implementation adapted to a specific filtering strategy.

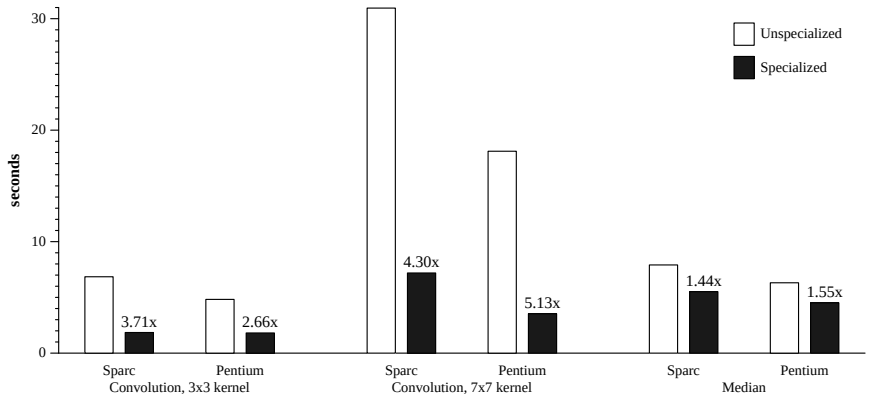

Figure 10: Prospective benchmarks. Manual back-translation from C to Java.

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 UltraSparc) 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 9.

We conduct experiments on a three-by-three blurring convolution filter, a seven-by-seven blurring convolution filter, and a median filter. The speedups are between 1.5 times and 4 times, 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.

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 on the same machines as were used in the previous experiment. Figure 10 shows the result of our experiments.

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 slower than the Harissa environment, memory access is not as dominating a factor as with the Harissa tests, making benefits of the code simplifications performed by the specializer more apparent.

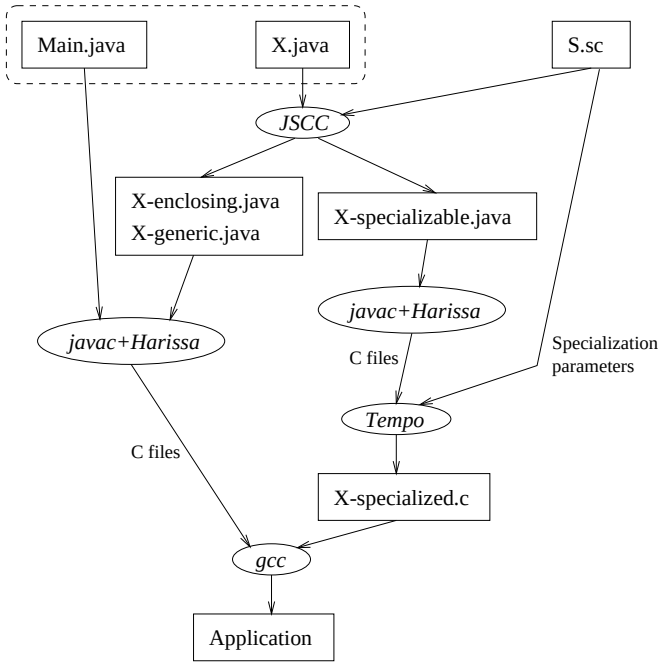

Figure 11: Structure of the prototype, specialization of class `X`

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 to expose specialization opportunities. Examples include virtual calls, casts, and array references (see Section 4). Thus, following our approach, the kinds of specialization that can be expressed using this approach are not limited by the syntax and semantics of Java, but by the expressiveness of C.

5.1 Structure of the Prototype

The overall structure of the prototype is presented in Figure 11. The input to the specialization process is a Java program and a set of specialization classes. The Java Specialization Class Compiler `JSCC` [29] 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 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. The latter option is the only one currently implemented. Note that Java programs can be specialized at both compile time and run time.

Specialization classes are fundamental to our approach: (i) they allow the programmer to concentrate on defining interesting optimization scenarios; (ii) they abstract over the optimization process, expressing what to optimize as opposed to how to optimize it. The former advantage allows the user to work at the level of the original source program, while the latter results in masking the numerous parameters of the specialization engine.

5.2 Harissa

Harissa is an off-line compilation system that fully 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 translator such as Tempo. Furthermore, Harissa has been extended to generate C code that contains extra information useful to Tempo.

5.3 Tempo

Tempo is an off-line program specializer for the C language [8], that supports both compile-time and run-time specialization. We made a few extensions to Tempo to more effectively treat 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 is different in C and in Java, all objects being dynamically allocated in the latter. 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 Java. 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. The existing related work includes a program specializer for a small language based on Emerald, and a formal treatment of the specialization of virtual calls. 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 both data representation and control flow specialization.

Specialization of object-oriented languages

Marquard and Steensgaard developed a partial evaluator for a small object-oriented language based on Emerald [20]. They focus 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 present partial evaluation as a means for eliminating virtual calls in simple object-oriented language [15]. Their work focuses on formalizing the analysis and transformations realized by performing program specialization on programs with virtual calls.

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 [21]. 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, as is discussed in Section 7. 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).

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. This technique is incorporated in the forthcoming HotSpot Java compiler.

7 Future Work

We have provided an outline of how 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.

First, we outline a major application domain: software components. Then, we discuss issues specific to producing pure Java programs rather than Harissa-specific programs. Finally, we investigate how the transformations performed by program specialization can be generalized to also include data representation.

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 highly-generic components, that once integrated specialize in a predictable way into efficient implementations.

Java as target language

For portability purposes, it can be beneficial to produce specialized code in Java. To do so, 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).

A translation back into Java is possible if the C specialized code has enough information and structure to recover Java constructions. Given our specialization process, this relies 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.

Data representation transformations

This paper concentrates on the optimization of the control flow that depends on static data. Specialization also exposes opportunities to optimize the data flow. Specifically, in the specialized image filtering example, the accesses through indirections to object fields are a key source for further optimization. Rather than manipulating data through object fields, it is more efficient to store data in local variables. These are cheaper to access, and trigger optimizations by a C compiler.

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, 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 included in 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.

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.J. Eggers, and B.N. Bershad. Fast, effective dynamic compilation. In PLDI'96 [25], 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 Loong A. 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] B. Guenter, T.B. Knoblock, and E. Ruf. Specializing shaders. In *Computer Graphics Proceedings*, Annual Conference Series, pages 343–350. ACM Press, 1995.

[15] S. C. Khoo and R. S. Sundaresh. Compiling inheritance using partial evaluation. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 211–222, Yale University, 17–19 June 1991.

[16] 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.

[17] P. Lee and M. Leone. Optimizing ML with run-time code generation. In PLDI'96 [25], pages 137–148.

[18] 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.

[19] 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.

[20] 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.

[21] B. Moura. *Bridging the Gap between Functional and Imperative Languages.* PhD thesis, University of Rennes I, April 1997.

[22] 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.

[23] 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.

[24] G. Muller and U.P. Schultz. Harissa: Efficient Java execution by global program optimizations. *IEEE Software*, 1999. To appear.

[25] *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996. ACM SIGPLAN Notices, 31(5).

[26] 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.

[27] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[28] J. C. Russ. *The Image Processing Handbook.* CRC Press, Inc., second edition, 1995.

[29] 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.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399