

Automatic Program Specialization for Java*

Ulrik Schultz
DAIMI, University of Aarhus
Aarhus, Denmark
ups@daimi.au.dk

Charles Consel
Compose Group,
LaBRI/ENSERB
Talence, France
charles.consel@labri.u-bordeaux.fr

November 20, 2000

Abstract

The object-oriented style of programming facilitates program adaptation and enhances program genericness, but at the expense of efficiency. We demonstrate experimentally that state-of-the-art Java compilation technology fails to compensate for the use of object-oriented abstractions to implement generic programs, and that program specialization can be used to eliminate these overheads. We present an automatic program specializer for Java, and demonstrate experimentally that significant speedups in program execution time can be obtained through automatic specialization. Although automatic program specialization could be seen as overlapping with existing optimizing compiler technology, we show that specialization and compiler optimization are in fact complementary.

1 Introduction

Object-oriented languages encourage a style of programming that facilitates program adaptation. Encapsulation enhances code resilience to program modifications and increases the possibilities for direct code reuse. Message passing lets program components communicate without relying on a specific

*Based on work done in the Compose Group at IRISA/INRIA, Rennes, France; supported in part by Bull.

implementation; this decoupling enables dynamic modification of the structure of the program in reaction to changing conditions. The use of these object-oriented abstractions in well-tested object-oriented designs (such as design patterns [11]) naturally leads to the development of generic program components.

Program genericness is however achieved at the expense of efficiency. Encapsulation isolates individual program parts and increases the cost of data access. Message passing is implemented using virtual dispatching; virtual dispatching not removed by a compiler can be a major overhead in the execution of a program. Due to the complex interaction that takes place between objects in a generic program part, virtual dispatching in generic code is inherently difficult to remove using compiler optimizations. A virtual dispatch obscures program control flow and blocks traditional hardware and software optimizations. In this paper, we experimentally demonstrate that state-of-the-art Java compiler technology fails to completely eliminate the overheads due to the use of object-oriented abstractions to implement generic programs.

The overheads due to genericness implemented using object-oriented abstractions can be eliminated using program specialization. Program specialization is a technique for adapting a program to a given execution context; when applied within the object-oriented paradigm, program specialization simplifies the interactions that take place between the program objects. This simplification is based on a global knowledge of these interactions. In this paper, we experimentally demonstrate that program specialization gives significant speedups when combined with state-of-the-art Java compiler technology.

We have developed an automatic program specializer for Java, named JSpec. JSpec is targeted towards optimizing realistic Java programs. It combines static analyses with aggressive global optimizations. JSpec automatically derives specialized programs that execute significantly faster than their generic counterparts. In this paper, we describe how JSpec specializes Java programs, and experimentally demonstrate that it automatically eliminates overheads due to the use of object-oriented abstractions in writing generic programs.

Earlier work has addressed the declaration of what to specialize, in the form of specialization classes [29], described an early prototype of JSpec [24], and addressed the issue of selecting where to specialize [25]. Here, we present the complete Java-to-Java specialization performed by JSpec, and experimentally demonstrate the advantage of program specialization on a wide selection of generic benchmark programs. Lastly, we here demonstrate that automatic program specialization does not overlap with existing optimizing compiler technology, but instead is complementary.

```

abstract class Binary {
    abstract int eval( int x, int y );
}
class Add extends Binary {
    int eval( int x, int y ) {
        return x+y;
    }
}
class Mult extends Binary {
    int eval( int x, int y ) {
        return x*y;
    }
}

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

```

Figure 1: Binary operators and a power function.

Overview

First, Section 2 describes how object-oriented programs can be automatically specialized, and outlines the JSpec implementation. Then, Section 3 investigates the relation between automatic program specialization and aggressive object-oriented compiler optimizations. Section 4 describes a set of benchmark programs and presents the result of applying automatic program specialization to these programs. Last, Section 5 discusses related work, and Section 6 presents our conclusion and discusses future work.

Terminology

The terminology used in the fields of object-oriented programming and program specialization overlap, which can lead to confusion. First, in some object-oriented languages, class fields and class methods are referred to as static fields and static methods. The word “static” is however used in program specialization to indicate information known during the specialization phase. To resolve the conflict, we never refer to class fields and class methods as static fields and static methods. Second, a subclass is often said to “specialize” its superclass. To avoid confusion, we refer to the relation between a subclass and its superclass in terms of inheritance (the subclass inherits from the superclass) or in terms of the subclass/superclass relation (one class subclasses some other class, or one class is the superclass of some other class).

Example

As an example of program specialization, Figure 1 shows a collection of four classes, `Binary`, `Add`, `Mult`, and `Power`. The class `Binary` defines a standard

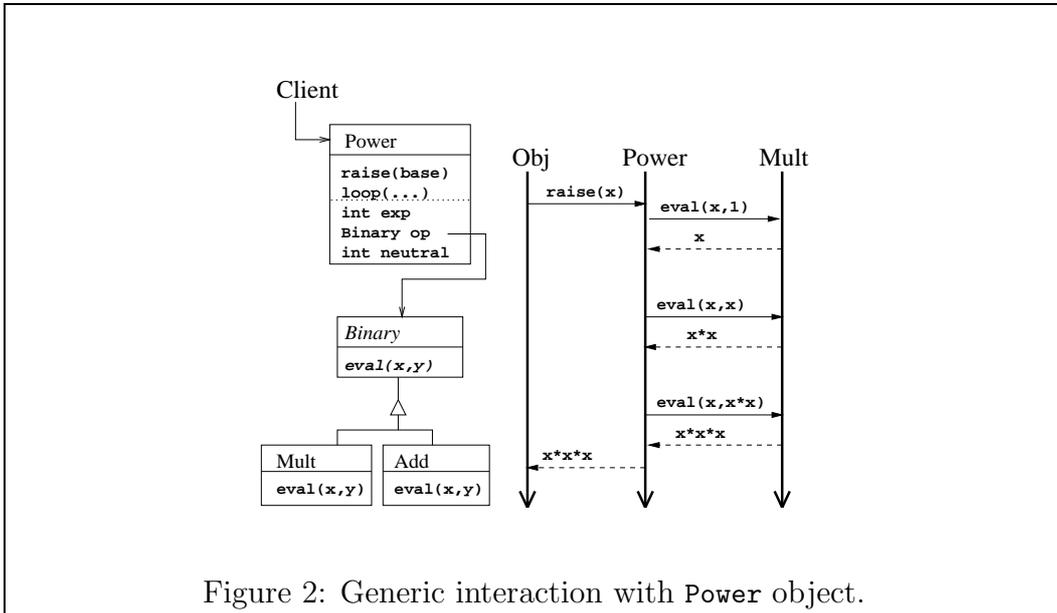


Figure 2: Generic interaction with Power object.

type for the concrete binary operators Add and Mult. The Power class can be used to apply a Binary operator a number of times to a base value. The Power class can be used as follows:

```
(new Power( y, new Mult(), 1 )).raise( x )
```

to compute x^y . The object diagram of this program is shown in Figure 2, along with the object interaction diagram that results from computing x^3 .

We can specialize this program for the object interaction that takes place

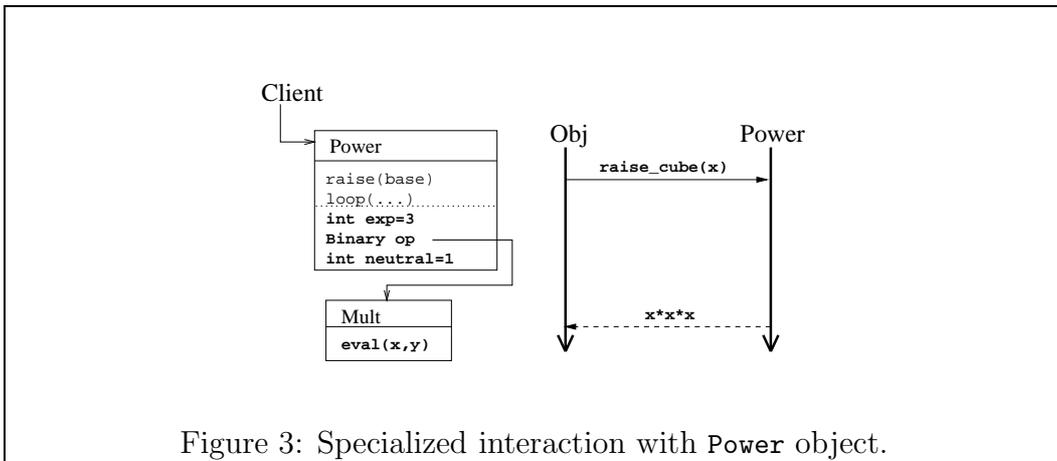


Figure 3: Specialized interaction with Power object.

when computing x^3 . Sending the message `raise(x)` to the `Power` object gives rise to a series of object interactions between the `Power` object and the `Mult` object that result in the return value `x*x*x`. To optimize for the case where the `Power` object is used to compute x^3 , we can enable this object to directly produce the result of `x*x*x`. Specifically, we can add a method

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

to the class `Power`, and any clients can use this new method to compute the result. The object state used for specialization and the resulting object interaction are shown in Figure 3. A client that is specialized with this state can use the `raise_cube` method to compute the result more efficiently. Automatic program specialization, as defined in the next section, can automatically derive such a specialized program. As shown in the experiments reported in Section 4, such simple specialization of the `Power` object produces a speedup of 4–7 times when compiled using state-of-the-art Java compilers.

2 Java Program Specialization

We now give an overview of automatic program specialization, explain how an object-oriented program can be specialized, and last describe the implementation of our automatic program specializer for Java.

2.1 Automatic program specialization

Program specialization optimizes a program fragment with respect to information about the context in which it is used, by generating an implementation dedicated to a given usage context. One approach to automatic program specialization is *partial evaluation*, which performs aggressive interprocedural constant propagation of *all data types*, and performs constant folding and control-flow simplifications based on the usage context. Partial evaluation adapts a program to known (*static*) information about its execution context, leaving behind only the program parts controlled by unknown (*dynamic*) data. Partial evaluation has been extensively investigated for functional [3, 5], logic [19], and imperative [1, 2, 6] languages. In this paper, we only consider off-line partial evaluation, where the program is first divided into static and dynamic computations using a *binding-time analysis*, and then specialized according to a concrete context by executing the static program parts on known data.

In contrast with most optimizing compilers, partial evaluation does not impose any bounds on the amount of computation that can be used to optimize a program. However, partial evaluation relies on the user to direct the specialization process towards the program parts which contains specialization opportunities. Furthermore, the user may have to control some transformations. Incorrect user guidance may result in over-specialization (code explosion) or under-specialization (no benefit from specialization). Because partial evaluation relies on highly accurate and complex analyses and the identification of specialization candidates, in practice it cannot be applied to complete programs. Instead, the strategy for applying this technique is usually to extract a specific slice from a program, specialize it, and then re-insert it into the original program [6]. An abstract description of the parts outside the program slice must be given to ensure that correct binding times are derived. For an object-oriented program, the program parts to specialize and a description of the specialization context can be declared using the *specialization class framework* [29].

2.2 Object-oriented program specialization

Let us now describe in detail how automatic program specialization transforms an object-oriented program. We first give a global overview of the specialization process, describe how specialization transforms the methods of the program, and address the issue of how to express the specialized program. Then, we discuss the features that are needed from a program specializer to effectively specialize object-oriented programs. Last, we discuss issues pertaining to specialization of object-oriented features in Java.

From a global point of view, the execution of an object-oriented program can be seen as a sequence of interactions between the objects that constitute the program. Fixing particular parts of the program context can fix certain parts of this interaction. Program specialization simplifies the object interaction as much as possible, by evaluating the static (known) interactions, leaving behind only the dynamic (unknown) interactions. We refer to program specialization for object-oriented languages as *object-oriented program specialization*.

Objects interact by using virtual calls to invoke methods. We can specialize the interaction that takes place between a collection of objects by specializing their methods for any static arguments. Methods are specialized with respect to the static part of their arguments (including the `this` argument), and are introduced in the original object under a new name. This transformation enables the object to respond to a new specialized message that other specialized methods can send to the object.

A method is specialized by optimizing any use of encapsulated values, any virtual dispatches, and any imperative computations:

Encapsulation: Data that are encapsulated inside an object can control computations elsewhere in the program. When such data are considered static by the specializer, they can be propagated to wherever they are used; computations that depend on these data can be reduced. Besides static computations, specialization also reduces the operation needed to access the data (an indirect memory reference).

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

Imperative computations: Methods are specialized using transformations common to imperative program specializers, such as constant propagation (of all data types), constant folding, conditional reduction, and loop reduction. In the Java specialization experiments reported in Section 4, most specialization scenarios require treating a mix of object-oriented and imperative constructs.

To complete the specialization process, the specialized program must be expressed as source code. The specialized methods must be added to the classes of the program. Introducing these methods directly into the classes of the program is problematic: encapsulation invariants may be broken by specialized methods where safety checks have been specialized away, and this mix of generic and specialized code obfuscates the appearance of the program and complicates maintenance. A representation of the specialized program that preserves encapsulation and modularity is needed.

The dependencies between the specialized methods follow the control flow of the program, which cuts across the class structure of the program. Aspect-oriented programming is an approach which allows logical units that cut

across the program structure to be separated from other parts of the program and encapsulated into an aspect [17]. We encapsulate the methods generated by a given specialization of an object-oriented program into a separate aspect, and weave the methods into the program during compilation. Access modifiers can be used to ensure that specialized methods can only be called from specialized methods encapsulated in the same aspect, and hence always are called from a safe context. Furthermore, the specialized code is cleanly separated from the generic code, and can be plugged and unplugged by selecting whether to include the aspect in the program.

A program specializer that targets realistic object-oriented programs must consider each use of an object individually when determining its binding time. Objects often act as basic entities (e.g., complex numbers) or as containers; restricting all objects of the same class to have the same binding time will often result in these objects being considered dynamic, which again results in under-specialization. Specifically, to ensure adequate binding times, a use of a static object in a dynamic context must not force this object to be considered dynamic; the data stored in each object instance must be given individual binding times; and, each invocation of a method must be considered individually.

A language like Java has no syntax for expressing direct calls to ordinary methods; a virtual dispatch simplified by specialization into a call to a single specialized method is expressed using a downwards type cast to the concrete type of the receiver object, and the specialized method is generated as a `final` method. Methods declared in an abstract class or in an interface can be specialized similarly to ordinary virtual methods; we refer to the first author's PhD dissertation [23] for details.

2.3 JSpec

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

- JSpec treats the entire Java language excluding exception handlers and `finally` regions. It takes as input either Java source code, Java bytecode, or native functions.
- JSpec is integrated with an extended version of the specialization class framework, which allows the binding times of the targeted program slice to be specified separately from the program using a declarative language.

- The JSpec binding-time analysis is context-sensitive, type-polyvariant (objects of same type are given individual binding times), use-sensitive [14], and flow-sensitive; object values are treated individually for each instance. This strategy ensures a consistent behavior that does not impose arbitrary limitations on the specialization process.
- Specialized programs are generated either as Java source code encapsulated in an AspectJ [30] aspect, C source code for execution in the Harissa [21] environment, or binary code generated at run time for direct execution in the Harissa environment.
- JSpec is applied to a user-selected program slice, which allows time-consuming analyses and aggressive transformations to be directed towards the critical parts of the program.

In this paper we only consider Java-to-Java specialization; we refer to earlier work [24] and the first author’s PhD dissertation [23] for more information on using C as input or output language.

JSpec has been designed with an emphasis on re-use of existing technology. In particular, JSpec uses a specializer for C programs, named Tempo [6], as its partial evaluation engine. This approach allows us to take advantage of the advanced features found in a mature partial evaluator. C is used as intermediate language in JSpec; the low-level nature of C makes it possible to represent the semantics of Java programs. A Java-to-C compiler, called Harissa [21], is used to translate Java programs into C. A dedicated back-translator maps the C representation of a specialized Java program back into Java. Finally, AspectJ is used to weave the specialized Java program with the generic program to produce a complete, specialized program.

Java features such as exceptions, multi-threading and dynamic loading must be taken into account. JSpec does not support exception handlers in the program slice being specialized, and can therefore consider `throw` statements to terminate the program. JSpec only specializes a single thread of control, and speculatively evaluates code inside synchronized regions; while not appropriate in all situations, this approach offers a simple and predictable behavior. Dynamic loading is implicitly handled since JSpec only specializes a given program slice: classes that may be dynamically loaded by the program must either be explicitly included in the program slice to be specialized, or considered external code and be abstractly described.

```

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

```

(a) Exponent, operator, neutral value static

```

aspect PowerOfTwo {
  introduction Binary {
    int eval(int x) { throw new JSpecExn(); }
  }
  introduction Add {
    int eval_2(int x) { return x+2; }
  }
  introduction Mult {
    int eval_2(int x) { return x*2; }
  }
  introduction Power {
    int raise_pow2() {
      int result = neutral, e = exp;
      while( e-- > 0 ) result = op.eval_2(result);
      return result;
    }
  }
}

```

(b) Base value static

Figure 4: Various specializations of Power.

2.4 The Power example revisited

Having outlined the specialization process, let us revisit the `Power` example of the introduction. In this example, the `Power` object is static, as is all of its fields. The virtual dispatches to the method `eval` are reduced, and the result can be expressed using AspectJ syntax. The result is shown in Figure 4a encapsulated in the aspect `Cube`. An aspect is a named list of `introduction` blocks; an `introduction` block lists a collection of methods to introduce into the class given at the head of the `introduction` block. Alternatively, we can specialize for the inverse scenario. When all fields of the `Power` object are dynamic and the parameter `base` is static, the `eval` methods are speculatively specialized for the `base` value, as shown in Figure 4b in the aspect `PowerOfTwo`. (The method introduced into the `Binary` class is needed to make a correct program, but is not created as an abstract method; indeed, using an abstract method would force any subclass outside the targeted program slice to implement this method.)

3 Specialization vs. Compiler Optimization

Automatic program specialization and compilers rely on similar optimizations. However, as we describe in this section, compilers perform optimizations not considered in automatic program specialization, and automatic program specialization performs optimizations beyond the capabilities of a compiler. Based on these observations, we argue that program specialization and compiler optimizations are complementary.

3.1 Common points

Automatic program specialization uses type information to eliminate virtual dispatches; this is a technique often employed by aggressive compilers for object-oriented languages. The most common techniques are *customization* [4] and the more general *selective argument specialization* [7]. Both automatic program specialization and selective argument specialization create new specialized methods by propagating type information about the `this` and the formal parameters of a method throughout the program, and introduce new specialized methods into the class of the corresponding unspecialized method.

3.2 Compiler optimization capabilities

Most compiler optimizations are targeted towards optimizing all parts of a program, and improving performance regardless of the usage scenario. In contrast, automatic program specialization only optimizes a specific aspect of a program (that depends on static parts of the context), and reproduce all other parts of the program verbatim. Therefore, program specialization is dependent on a compiler for traditional intra-procedural optimizations such as copy propagation, common subexpression elimination, loop invariant removal, etc. that are essential for performance. Furthermore, optimizations that are not expressible at the language level, such as register allocation and (e.g., in Java) array bounds check elimination, cannot be performed by the program specializer, and must be handled by the compiler.

Customization and selective argument specialization can both optimize methods for type information even when no information can be statically deduced about their type. Profiling can be used to gather information about the types of objects often passed to a given method. Multiple versions of the method can be generated, each specialized to one of these types. On the contrary, although program specialization can speculatively specialize the receivers of a virtual dispatch with a dynamic `this`, these receiver methods are not specialized for the `this`: the `this` is dynamic and thus not available during specialization.

3.3 Program specialization advantages

Program specialization allows the user to control the optimization process by supplying program configuration parameters. User control allows overheads to be targeted that are not normally eliminated by an optimizing compiler. For example, overheads that are distributed across the entire structure of a program and are hard to detect using automated profiling techniques can be eliminated using program specialization.

Compared to a compilation technique such as selective argument specialization, program specialization is more thorough, more aggressive, and more pervasive. Program specialization is more thorough because it propagates known values of any kind (even completely or partially known objects) throughout the program. This propagation is not only done through formal parameters and local variables, but also through object fields. Program specialization is more aggressive because it reduces any computation that is based solely on known information. No resource bound restricts the amount of simplification to be performed. Program specialization is more pervasive than aggressive compilation because it uses global knowledge about the ob-

ject behavior and object structure to propagate information globally and to reduce computations wherever possible.

The experiments reported in the next section clearly show that program specialization enhances the performance of programs beyond the capabilities of state-of-the-art Java compilers that employ profile-directed optimizations.

3.4 Specialization and compiler optimization working together

There is an overlap between object-oriented program specialization and object-oriented compiler optimizations: both aim to eliminate virtual dispatching to enhance the performance of the program. Nevertheless, when the critical parts of a program are highly polymorphic in their use of different objects, an optimizing compiler cannot easily determine the object interaction, and will fail to generate efficient code. In this case, automatic program specialization can be supplied information about those parts of the execution context that control object interaction at critical program points, and then be used to simplify the object interaction at these points. The resulting program has a simpler object interaction, which makes it easier to use profile information to determine how the remaining parts of the program can be optimized. When the behavior of a program part changes in unpredictable ways, or when program specialization cannot determine that certain program parts behave in a fixed way, profile-guided optimization may still be able to optimize the program.

4 Experimental Study

In this section, we compare the execution time of generic programs to the execution time of specialized programs. With the exception of the benchmark programs classified as “imperative,” the programs are written using object-oriented abstractions where appropriate. The programs are compiled using a selection of state-of-the-art Java compilation systems. Our goal is to show that the specialized programs execute faster than their generic counterparts. If the specialized programs execute faster, then JSpec and the optimizing compilers do not overlap; that is, the genericness of the benchmark programs is not (or is partially) eliminated by the optimizing compilers.

4.1 Benchmark programs

To properly assess the performance improvements due to partial evaluation, we have considered a wide selection of generic benchmark programs. The programs are grouped by the primary kind of specialization opportunity they expose, namely *imperative*, *object-oriented*, and *mixed*. The programs are written in a generic programming style; this design decision is an advantage in terms of software engineering, but is a disadvantage in terms of performance. All benchmark programs are computationally intensive. They do not perform a large amount of I/O, do not allocate a large amount of memory, and do not contain multi-threaded code. In all of the benchmark programs, we have avoided the use of access modifiers, to enable JSpec to perform uninhibited inlining as a post-specialization optimization; although unrealistic, this choice allows us to measure the benefit due to the inlining that can be performed by JSpec.

Imperative opportunities

Some object-oriented programs are primarily imperative in nature, although they may benefit from object-oriented constructs to provide structuring or data encapsulation. The benchmark programs that exhibit imperative opportunities for specialization could have been written in an imperative language and then specialized using a partial evaluator for this language. However, little is known about the efficiency of imperative Java programs after partial evaluation; therefore, we consider it interesting to include such programs in our benchmark suite. In each program, base-type data in the execution context is static.

FFT: The Fast-Fourier Transform benchmark taken from the Java Grande benchmark suite [9]. The radix size is static, and the data being transformed is dynamic. Specialization is done for three different radix sizes, 16, 32, and 64.

Power-1: A standard version of the power function, written using a loop and with a fixed operator (multiplication) and neutral value. The exponent is static, and the base value being raised is dynamic. Specialization is done for an exponent value of 16. (This benchmark is a classical partial evaluation example.)

Compilers normally do not perform aggressive optimizations over base-type values, and thus cannot perform optimizations similar to those performed by program specialization.

Object opportunities

The adaptive behavior of some object-oriented programs is completely controlled through object-oriented mechanisms; such programs can be said to provide pure object-oriented specialization opportunities. Simplification of virtual dispatches without taking imperative features such as loops, conditionals, and other computations into account is sufficient for specializing these programs. In each of these programs, the object composition is fixed in the program, and the program can be optimized without the use of additional configuration information.

Builder: Matrices with a dense or sparse representation are created using the builder design pattern and subsequently exponentiated. The choice of builder pattern is static, and the matrix dimensions and contents are dynamic.

Bridge: A Mandelbrot fractal is computed using complex numbers. The complex numbers are separated into an interface and an implementation (cartesian or polar coordinates) using the bridge design pattern. The bridge coupling is static, and the actual complex numbers manipulated are dynamic.

Iterator: A set data structure is implemented over an underlying primitive data structure (array or linked list). The iterator design pattern is used in the implementation of member and intersection operations. The choice of underlying data structure is static, and the data being manipulated is dynamic.

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

Mixed opportunities

In many object-oriented programs, the adaptive behavior is controlled by a mix of object-oriented mechanisms and imperative constructs. In the programs exhibiting “mixed” specialization opportunities, specializing the program to a static input requires treating a mix of object operations and imperative computations. In each program, both base-type data and object data in the execution context is static.

ArithInt: A simple arithmetic expression interpreter, used to compute the maximal value of a function supplied as data. The arithmetic expression is static, and the contents of the environment is dynamic. Specialization is done for a function mapping integer planar coordinates into an integer value.

ChkPt: The checkpointing example of Lawall and Muller [18]: a generic checkpointing routine for a binding-time analysis implementation is specialized to object composition properties specific to each phase of the binding-time analysis. Specialization and benchmarking are done for the binding-time analysis phase only.

Fold: A binary operator folded over a list. In experiment **LS**, the list contents are static, and the operator and initial value are dynamic. In experiment **OP**, the operator and initial value are static, and the list contents are dynamic. Specialization is done for a list of length 50 and a multiplication operator.

IP: An generic image filtering program, where the image representation and the filter to apply are abstracted using design patterns [24]. Specialization is done for blurring convolution filters of size 3×3 and 5×5 .

Pipe: Simple mathematical functions composed together to form a pipe, applied to a single input value. The function composition is static, and the value input to the function pipe is dynamic.

Power-2: The power example from Section 1, with the exponent, operator and neutral values as static, and the base value as dynamic. Specialization is done for a combination of two objects of class **Power** with different operators.

Strategy: A number of single-pixel image operators (e.g., change pixel brightness) encapsulated into separate algorithms using the strategy design pattern are applied to an image. The choice of operators is static, and the image data is dynamic.

We expect that program specialization will improve the efficiency of these programs, since the program will be simplified beyond what we can expect from ordinary compiler optimizations.

4.2 Methodology

Experiments were performed on two different machines, a SPARC and an IA32. The SPARC machine is a Sun Enterprise 250 running Solaris 2.7, with two 300MHz Ultra-SPARC processors (all benchmarks are single threaded). The IA32 machine is a Dell OptiPlex GX1 running Linux 2.2, with a single 600MHz Pentium III processor.

All benchmarks are automatically specialized using JSpec. Compilation from Java source to Java bytecode is done using Sun's JDK 1.1 javac compiler with optimization selected (JSpec by default uses Sun's JDK 1.1 tools to produce a set of specialized class files). We use JIT, dynamic and off-line compilers to compare the performance before and after specialization.

The JIT benchmarks are performed on SPARC using Sun’s JDK 1.2.2 JIT compiler [26], and on IA32 using IBM’s JDK 1.3 JIT compiler [15]. The dynamic compilation benchmarks are performed using the HotSpot compiler included with Sun’s JDK 1.3 beta 2 [27] running in “server” compilation mode; HotSpot is available both for SPARC and IA32. The off-line compilation benchmarks are done using the Harissa bytecode compiler [21], with optimization level **E03**, except for the **ChkPt** benchmark where optimization level **E01** was selected to limit resource consumption during compilation. The maximal heap size was set to 96Mb for all systems except Harissa (Harissa does not provide any means of limiting the amount of memory allocated by the program).

Each benchmark is structured as follows: a benchmark performs ten main iterations that are timed individually, and each such main iteration consists of some number of minor iterations of the actual test. The first five main iterations are discarded to allow JIT and dynamic compilers to optimize the program. All execution times are reported in seconds, and the number of minor iterations is adjusted to ensure that each main iteration runs long enough to give consistent time measurements. All benchmarks compute and print a checksum value which is threaded through the computation of each iteration of the benchmark, to prevent compiler optimizations such as loop invariant removal or dead code elimination from removing the code that is being benchmarked.

The current implementation of JSpec does not automatically generate the code needed for transparent reintroduction of specialized code into a program. For this reason, the call to the specialization entry point is manually created. All benchmarks except **ChkPt** are automatically specialized; due to implementation limitations, the **ChkPt** benchmark requires patching after specialization.

4.3 Results

With the current implementation of JSpec, specialization always increases program size, since new methods are added to the program and no methods are removed. The size of the program slice targeted with JSpec ranges from roughly 10 lines (**Power-1**) to roughly 1100 lines (**ChkPt**) of Java source code. We measure the size increase due to specialization as the ratio between the size of the unspecialized program slice and the resulting AspectJ aspect. In the **FFT** and **Fold:LS** benchmarks there is a large size increase of the program code due to specialization: in **FFT:64** there is an 11-times size increase, and in **Fold:LS** there is a 40-times size increase. Across all other benchmarks, there is an average 2.8-times size increase due to specialization.

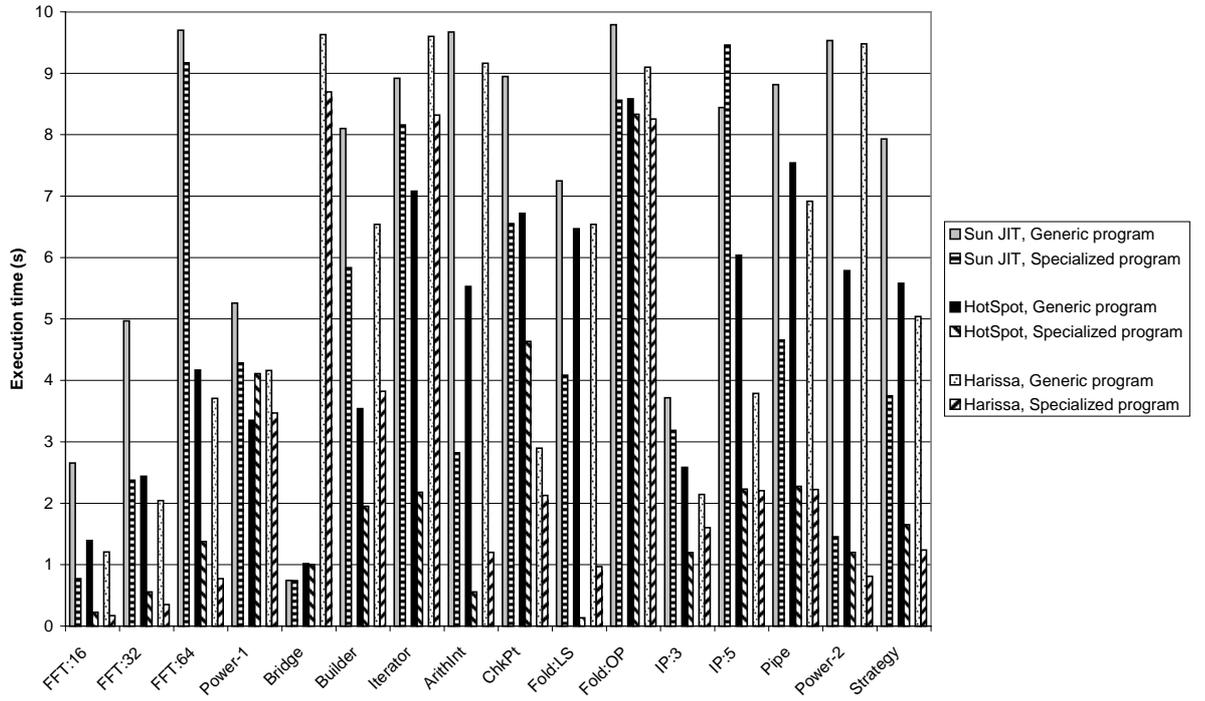


Figure 5: Execution time of generic and specialized program, SPARC.

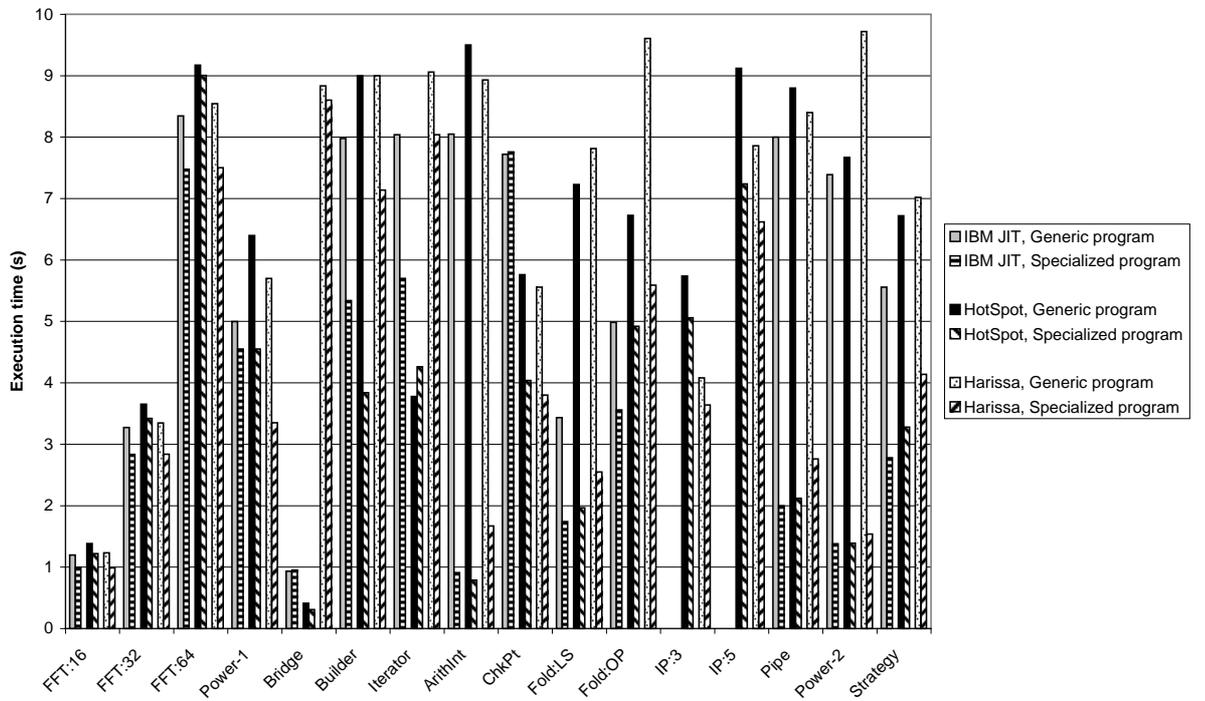


Figure 6: Execution time of generic and specialized program, IA32.

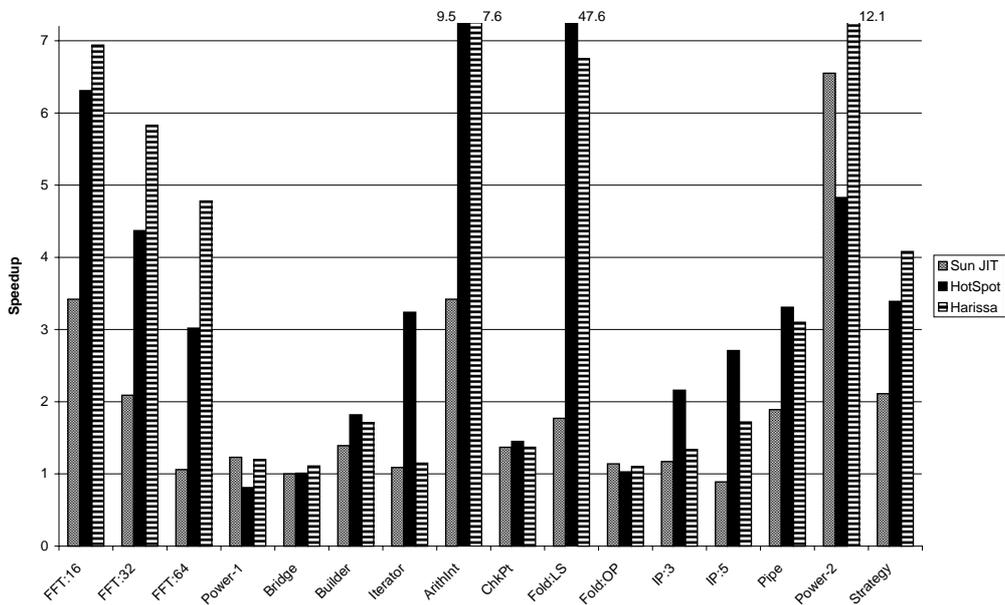


Figure 7: Speedup comparison (generic time/specialized time), SPARC.

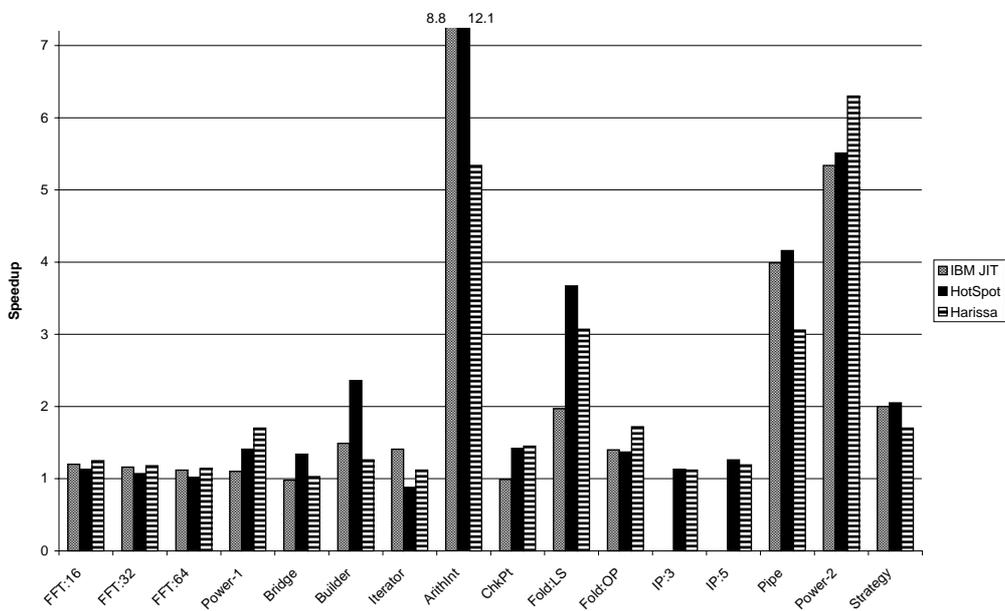


Figure 8: Speedup comparison (generic time/specialized time), IA32.

The execution time variation due to specialization is shown in Figures 5 (execution on SPARC) and 6 (execution on IA32). Execution times have been normalized between 0 and 10 seconds. The speedup due to specialization is shown in Figures 7 (execution on SPARC) and 8 (execution on IA32). Here, the execution time of each generic program is compared to the execution time of the specialized program; a result of 1.0 means that there is no speedup due to specialization. Results are shown only in the form of bar charts; we refer to the first author’s PhD dissertation [23] for tables with absolute execution times. No results are shown for IBM’s JIT on the IP benchmarks, since a runtime error was (incorrectly) generated when running this benchmark on this system.

On both architectures, there are high speedups for the **ArithInt**, **Fold:LS**, **Pipe**, **Power-2**, and **Strategy** benchmarks. Furthermore, on SPARC, there are high speedups for the **FFT**, **Iterator** (HotSpot only), and **IP** benchmarks. The high speedup for the **Fold:LS** benchmark when executing with HotSpot on SPARC is due to arithmetic simplifications carried out by the compiler; although a correct benchmark, we consider the speedup to be artificially high. Moreover, on SPARC there are significant speedups for **Builder** and **ChkPt**, and on IA32 there are significant speedups for **Builder**, **Iterator**, and **ChkPt**.

The results of performing JSpec inlining optimization after specialization is not illustrated in the figures (we again refer to the first author’s PhD dissertation for details). JSpec inlining optimization provides a significant benefit in some cases with Sun’s JIT and in a few cases with Harissa on IA32. However, inlining optimization is often detrimental to performance, especially on HotSpot. Indeed, JSpec inlining optimization is never an advantage on HotSpot.

4.4 Assessment

We observe that the average speedup gained by specialization across all compilers is 3.0 on SPARC and 2.4 on IA32 (averages are shown in Table 1). We attribute the greater benefit on SPARC to the fact that the SPARC is a RISC architecture, and thus to a higher degree depends on the compiler to optimize the program before execution.

As for the selection of compilers included in the study, on SPARC there is a significant difference between the execution time average for Sun’s JIT and the execution time averages for HotSpot and Harissa. On IA32, there is little difference between execution time averages. Looking closer at the results for SPARC, it appears that the more aggressive the compiler, the greater the benefit from specialization. However, a larger selection of Java compilers would be necessary to completely study the relation between the

	SPARC	IA32	average
JIT	2.0	2.4	2.2
HotSpot [†]	3.3	2.6	3.0
Harissa	3.8	2.1	3.0
average	3.0	2.4	2.7

[†]: excluding Fold:LS on SPARC

Table 1: Average speedups from Figures 7 and 8.

aggressiveness of compiler optimizations and the speedup due to specialization.

We conclude that JSpec inlining optimization is unnecessary in most scenarios. Programs can be specialized effectively without applying inlining optimization, which avoids problematic issues related to access modifiers when generating the specialized program as source code. After specialization the compiler can perform inlining optimization on the generated program without any concern for access modifiers, and can even take the configuration of the target machine into account.

5 Related Work

The optimization techniques most directly related to our work are customization and selective argument specialization; their comparison to program specialization has been detailed in Section 3. In essence, program specialization is more thorough, more aggressive, and more pervasive than these techniques. Static analyzes such as precise type inference [22] or CHA [8] can be used to eliminate virtual dispatching, but only when there is a single possible receiver; in highly generic code, there typically are multiple receivers for virtual dispatches. Nonetheless, a virtual call can be replaced with an explicit selection of which callee to invoke with a direct call [12, 13]. This highly aggressive optimization must be guided with profile information to avoid code explosion, and retains the cost of a runtime decision.

The rest of this section covers an automatic program specializer for Emerald, an automatic program specializer for C++, and a discussion of the intrinsic specialization model built into the C++ language.

Program specialization for object-oriented languages can be done using on-line [16] specialization techniques, as shown by Marquard and Steens-

gaard [20]. They developed a program specializer for a subset of Emerald — an object-based language. However, this work focuses on on-line specialization. There are no consideration on how program specialization should transform an object-oriented program, and virtually no description of how their program specializer handles object-oriented language features.

Program specialization can be done based on constructor parameters at run time for C++ programs, as shown by Fujinami [10]. Annotations are used to indicate member methods that are to be run-time specialized; a method is specialized using standard program specialization techniques for C and by replacing virtual dispatches through static object references by direct method invocations. Furthermore, if a virtual method invoked through a static object reference has been tagged as `inline`, it is inlined into the caller method and specialized. This approach to program specialization for an object-oriented language concentrates on specializing individual objects. On the contrary, we specialize the interaction that takes place between multiple objects based on their respective state, resulting in global program specialization.

Templates in C++ can be used to perform program specialization at compile time, as demonstrated by Veldhuizen [28]. By using a combination of template parameters and C++ `const` constant declarations, arbitrary computations over base type values can be performed at compile time. Compared to our definition of program specialization for object-oriented languages, specialization with C++ templates is limited in a number of ways. First, the values that can be manipulated are more restricted; for example, objects cannot be dynamically allocated. Second, the computations that can be simplified are more limited; for example, virtual dispatches cannot be simplified. Last, an explicit two-level syntax must be used to write programs; as a consequence, binding-time analysis must be performed manually, and functionality must be implemented twice if both a generic and a specialized behavior is needed.

6 Conclusion and Future Work

The development of generic software components is a growing trend in software development; however, this trend comes at the expense of performance. We argue that program specialization is a key technology to reconcile genericness and performance in that it enables a single component, designed to handle a whole family of problems, to be instantiated into an efficient implementation.

In this paper, we identify the overheads introduced by the development of generic object-oriented programs. We demonstrate that these overheads

can be automatically eliminated by program specialization, and present the techniques used to specialize object-oriented programs. To validate our approach in practice, we have developed a program specializer for Java, named JSpec. Finally, we characterize the benefits of program specialization by conducting an experimental study on a variety of Java programs. This study shows that not only can JSpec produce a speedup of 2.7 on average but it is complementary to optimizing Java compilers.

Our next step is to tackle run-time specialization for Java. This form of specialization allows specialized programs to be generated at run time, with respect to values unavailable at compilation time. Run-time specialization could be done by dynamically generating bytecode encapsulated into classes; this strategy would preserve the platform-independence feature of Java. However, the specialization principles presented in this paper rely on adding new methods to existing classes, which cannot be done at run time in Java. Changes to our specialization approach are needed to enable run-time specialization while complying with the restrictions of the Java virtual machine.

Availability

JSpec will be made publicly available at
<http://www.irisa.fr/compose/jspec>

References

- [1] L. 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] 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 (PEPM'94)*, pages 119–132, Orlando, FL, USA, June 1994. Technical Report 94/9, University of Melbourne, Australia.
- [3] A. Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.
- [4] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, A dynamically-typed object-oriented programming

- language. In B. Knobe, editor, *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation (PLDI '89)*, pages 146–160, Portland, OR, USA, June 1989. ACM Press.
- [5] C. Consel. A tour of Schism: a partial evaluation system for higher-order applicative languages. In *Partial Evaluation and Semantics-Based Program Manipulation (PEPM'93)*, pages 66–77, Copenhagen, Denmark, June 1993. ACM Press.
- [6] C. Consel, L. Hornof, F. Noël, J. Noyé, and E. 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, Feb. 1996.
- [7] J. Dean, C. Chambers, and D. Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI'95)*, pages 93–102. ACM SIGPLAN Notices, 30(6), June 1995.
- [8] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. G. Olthoff, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Århus, Denmark, Aug. 1995. Springer-Verlag.
- [9] J. G. Forum. The Java Grande Forum benchmark suite, 1999. Accessible from <http://www.javagrande.org> and <http://www.epcc.ed.ac.uk/javagrande>.
- [10] N. Fujinami. Determination of dynamic method dispatches using runtime code generation. In X. Leroy and A. Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation (TIC'98)*, volume 1473 of *Lecture Notes in Computer Science*, pages 253–271, Kyoto, Japan, Mar. 1998.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [12] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-guided receiver class prediction. In *OOPSLA '95 Conference Proceedings*, pages 108–123, Austin, TX, USA, Oct. 1995. ACM Press.

- [13] 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.
- [14] L. Hornof, J. Noyé, and C. Consel. Effective specialization of realistic programs via use sensitivity. In P. Van Hentenryck, editor, *Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, volume 1302 of *Lecture Notes in Computer Science*, pages 293–314, Paris, France, Sept. 1997. Springer-Verlag.
- [15] IBM. IBM JDK 1.3, 2000. Accessible from <http://www.ibm.com/java/jdk>.
- [16] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.
- [17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Longtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.
- [18] J. Lawall and G. Muller. Efficient incremental checkpointing of Java programs. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 61–70, New York, NY, USA, June 2000. IEEE.
- [19] J. Lloyd and J. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
- [20] M. Marquard and B. Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, University of Copenhagen, Apr. 1992.
- [21] G. Muller and U. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, pages 44–51, Mar. 1999.
- [22] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *OOPSLA'94 Conference Proceedings*, volume 29:10 of *SIGPLAN Notices*, pages 324–324. ACM Press, Oct. 1994.

- [23] U. Schultz. *Object-Oriented Software Engineering Using Partial Evaluation*. PhD thesis, University of Rennes I, Dec. 2000. Forthcoming.
- [24] U. Schultz, J. Lawall, C. Consel, and G. Muller. Towards automatic specialization of Java programs. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 367–390, Lisbon, Portugal, June 1999.
- [25] U. Schultz, J. Lawall, C. Consel, and G. Muller. Specialization patterns. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)*, pages 197–206, Grenoble, France, Sept. 2000. IEEE Computer Society Press.
- [26] Sun Microsystems, Inc. Sun JDK 1.2.2, 1999. Accessible from <http://java.sun.com/products/j2se>.
- [27] Sun Microsystems, Inc. Sun JDK 1.3, 2000. Accessible from <http://java.sun.com/products/j2se>.
- [28] T. Veldhuizen. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'98)*, pages 13–18, San Antonio, TX, USA, Jan. 1999. ACM Press.
- [29] E. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. In *OOPSLA'97 Conference Proceedings*, pages 286–300, Atlanta, GA, USA, Oct. 1997. ACM Press.
- [30] AspectJ home page, 2000. Accessible as <http://aspectj.org>.