A Machine Code Model for Efficient Advice Dispatch

Ryan M. Golbeck Gregor Kiczales
University of British Columbia
{rmgolbec,gregor}@cs.ubc.ca

Abstract

The primary implementations of AspectJ to date are based on a compile- or load-time weaving process that produces Java byte code. Although this implementation strategy has been crucial to the adoption of AspectJ, it faces inherent performance constraints that stem from a mismatch between Java byte code and AspectJ semantics. We discuss these mismatches and show their performance impact on advice dispatch, and we present a machine code model that can be targeted by virtual machine JIT compilers to alleviate this inefficiency. We also present an implementation based on the Jikes RVM which targets this machine code model. Performance evaluation with a set of micro benchmarks shows that our machine code model provides improved performance over translation of advice dispatch to Java byte code.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Code Generation, Optimization, Run-time environments

General Terms Languages, Performance

Keywords AspectJ, Aspect-oriented programming

1. Introduction

One of the design goals of ajc [16], a mainstream AspectJ [19] compiler, is to produce no performance overhead over a straight forward hand-coded implementation of the same functionality, as demonstrated by this quote from the AspectJ frequently asked questions:

> We aim for the performance of our implementation of AspectJ to be on par with the same functionality hand-coded in Java. Anything significantly less should be considered a bug [11].

ajc and other byte code rewriting based implementations [22, 23] face inherent performance limitations that stem from the semantics of Java byte code. There are a number of static invariants that are provably true in the AspectJ semantics that cannot be expressed in Java byte code.

Although byte code rewriting implementations have been critical to the adoption of AspectJ, and were the right strategy to employ at the time, the inherit expressibility limitations prevent them from surpassing the design goal quoted above.

Workshop VMIL '07, March 12-13, 2007 Vancouver, British Columbia, Canada

In this paper we present a runtime architecture for the AspectJ language which addresses these performance problems by binding of advice to join point shadows in machine code. We refer to our implementation as the AJVM. The contributions of this work are:

- a discussion of the limitations of Java byte code semantics in efficiently expressing AspectJ language constructs;
- a layout for a runtime architecture supporting late-binding of advice to join point shadows;
- a target machine code model for efficient execution of advice; and
- an implementation of this target model together with an evaluation and demonstration of the performance benefits of late-binding advice to join point shadows.

Our implementation achieves better performance than existing implementations of the AspectJ language semantics [6, 24, 16, 22]. It also meets or exceeds the performance of hand-coded comparable functionality, thereby satisfying the performance goals outlined in the AspectJ FAQ quoted above.

This paper is organized as follows. Section 2 discusses byte code rewriting based AspectJ implementations and briefly touches on the problems this strategy causes. Section 3 presents work on virtual machines supporting object-oriented programming and where in the architecture of these systems we can add functionality to enhance support for aspect-oriented (AO) programming [18].

Section 4 presents the primary results of the paper: a high level overview of the target machine code model together with the optimizations available to Just-in-Time (JIT) compilers, but not to byte code translators, that allow us to efficiently express AspectJ advice semantics.

This discussion is followed by the particulars of our implementation and the target machine code model for Intel i386 based processors in section 5. The results of our evaluation and benchmarks are presented in section 6. In sections, 7, 8 and 9 we present related work, future work and conclusions.

2. Byte Code Rewriting

The primary implementations of AspectJ to date are based on a compile- or load-time weaving process which operates on and produces Java byte code.

Compile- and load-time weaving in ajc are essentially the same. Compile-time weaving occurs during translation from AspectJ code to Java byte code, and load-time weaving is a byte code to byte code translation during the loading of the class. Both of these approaches are essentially byte code rewriting techniques. Load-time weaving has more conceptual similarities to the VM-based AO architecture we are developing, so we will draw our parallels to ajc's load-time weaver.

When using ajc's load time weaver, program compilation translates aspects into annotated class files and produces an XML as-

pect configuration file which describes the aspects that need to be loaded. When a program is executed, the ajc runtime is responsible for consulting the aspect definition file and loading the relevant aspects before classes are loaded; a customized class loader then weaves defined advice into the class byte code during loading.

Operating in this fashion, the static compilation phase of the program compiles aspects, but does not weave them. It produces an appropriate external representation of aspects, together with relevant meta-data, to be loaded by the runtime architecture. When the program is executed, the ajc runtime loads the external format into an internal representation and weaves class methods as they are loaded for advice dispatch.

These byte code rewriting implementations typically compile aspects into Java classes, compile advice bodies into methods associated with the aspect class, and arrange for advice to be executed by weaving method calls at advised join point shadows. This translation throws away information unique to aspects; after weaving, an advice body execution is no longer distinguishable from a normal method call. This loss of semantic information has a performance impact for advice execution, because, as we show below, we lose the ability to prove some static properties of the code.

3. Virtual Machine Architecture

An efficient VM based architecture supporting AO programming is analogous to studied architectures supporting object-oriented (OO) programming. Advice dispatch in an AO program parallels method dispatch in an OO program, and can be supported by a similar sort of architecture.

Like AO programming, some OO programming language implementations started out as pre-processing techniques. As OO techniques became more studied and accepted, these implementations were integrated into their own language compilers and eventually into virtual machine based runtime architectures.

Early VM-based OO architectures [12, 10] had five basic parts: an external format for representing classes that can be loaded into the runtime environment (external representation), an internal format for representing classes in the environment (internal representation), a dispatcher which determines the appropriate methods to be executed at a call-site (dispatcher), a format for laying out compiled code produced by a JIT compiler (target model), and a memory management scheme (memory manager). Note, however, that both the target model and memory manager are optional components. The target model can be omitted in an interpreter based implementation, and the memory management burden can be left to the programmer.

A VM-based environment can integrate additional components into each of these 5 parts of the architecture to provide a VM-based AO environment: additional external and internal representations for aspects, additional dispatch logic for advice execution, a modified layout for compiled code, and special rules for handling aspect instances in memory.

However, there is a rich history of work in VM-supported OO architectures [20, 17] since this initial basic design. Improvements include dynamic optimization compilers with adaptive optimization systems which control the JIT and its decisions during optimization that trade-off compile time and runtime efficiency of produced code. These systems use profiling data accumulated during the running of the program. Some VMs also have the ability to store profiling data on disk for cross-run persistence [2].

We present a VM-based AO implementation architecture that moves AspectJ along the path from language compiler, byte code based implementations to integration into a 5 part VM-based architecture. Our architecture mirrors the progression of previous OO architectures and follows in the foot steps of other VM-based AOP implementations such as PROSE[24], AspectWerkz[6] and Steamloom[4]. Our approach is conceptually similar to ajc load-time weaving: AspectJ code is compiled to Java byte code statically but using a compiler which preserves aspect specific information, and aspects are loaded and their advice executions woven at runtime. The primary difference between our implementation and traditional approaches for the purposes of this paper is that weaving is deferred from load time until JIT time. Waiting until JIT time allows improved performance by making it possible to exploit optimizations that are expressible in machine code, but not in Java byte code.

This approach also appears to enable a revised language semantics in which aspects can be loaded and unloaded dynamically. This paper does not advance AspectJ and AO programming all the way to a full VM-based implementation, it is one step along the path to realising such an implementation. Specifically, we do not integrate advice dispatch into the dynamic optimization system – instead, we weave advice as method calls, and rely on the existing profiling system to continue work as if the running program were purely OO-based.

4. Target Machine Code Model for Dispatch

In this section we elaborate on the mismatches between AspectJ and Java byte code semantics, and present the optimizations available in the JIT that alleviate the resulting inefficiency. The target machine code model[1] presented has few and tightly controlled coupling to the runtime architecture, thereby placing few restrictions on the design space of the supporting infrastructure. This model provides the flexibility required to design the remainder of the architecture with respect to other trade-offs in the VM, such as pointcut matching, class and aspect loading and unloading, and memory management techniques (garbage collection). These are factors that affect the overall runtime performance of a program and the runtime environment without specifically affecting the execution speed of the actual program code itself.

4.1 Common Case Optimizations

Because ajc compiles advice dispatch into Java byte code, it cannot take advantage of static invariants that are true during execution of the program. Consider a simple `before` advice on a dynamic call join point[2] defined in an `issingleton` aspect. At the Java Language Specification [13] level, the semantics of this advice dispatch could be implemented by adding a call such as the following to the join point shadow:

```
A.aspectOf().before$0();
```

ajc produces this equivalent byte code

```
//Method Aspect.aspectOf:()LAspect;
9:  invokestatic    #31;

//Method Aspect.before$0()V
12:  invokevirtual   #34;
```

Walking through this byte code we can see that it does more work than is required, and there are several possibilities for optimization if the target model is native machine code.

The first task is retrieving the aspect instance (byte code 9). This look-up, together with its implicit semantics, is a primary source of overhead in advice execution in ajc because the only ways to get an instance of an object in Java byte code are via `new`, to retrieve the

[1] In the remainder of this paper we use target model and target machine code model interchangeably.

[2] In the remainder of the paper we will use the more concise *join point* to mean dynamic join point.

instance from a variable, or to have it returned from some other method call. In this case, ajc retrieves the aspect using a static method call.

The body of the `aspectOf()` method is a reference to a static variable. We can reasonably expect a JIT compiler to inline the `aspectOf()` call because it is static and short. However, in the best case, inlining `aspectOf()` still results in a static variable reference which for typical VM memory management approaches becomes a constant load from memory in machine code.

However, because of the semantics of `issingleton` aspects, it is often possible to know the exact location in memory where the aspect instance is stored. This depends on the particulars of the memory management component of the architecture, but memory managers typically have a region of dedicated unmovable objects whose lifetimes correspond to the lifetime of the program [3]. By arranging to have the aspect instance allocated in this region, we can load it with one native instruction which loads the address of the instance as an immediate constant.

The next instruction, byte code 12, is composed of three different operations. The first performs a null check on the instance that was returned by `aspectOf()`. This is necessary because there is no way to express the fact that the aspect is guaranteed to exist. An aspect-aware JIT can eliminate this test because the aspect loader can guarantee that the aspect instance has been created before any advice is invoked.

Second, it causes a load from the aspect class' virtual function table. This load can also be eliminated by an aspect-aware JIT, because an aspect-aware VM can guarantee the exact type of the aspect corresponding to a specific advice call. This guarantee is sound because it is the aspect that arranges for its advice to be called, it is not a normal virtual method call.

The last operation is the actual call instruction which invokes the synthetic method that holds the body of the applicable advice. This call is necessary in any implementation for the advice body to be executed at all. Since advice bodies are compiled into standard methods internally, this advice call can be inlined like any standard method call. However, the runtime type of an instance cannot be proven at compile time; hence, when inlining the advice call a standard JIT compiler must place guards around the inlined method to check that the runtime type corresponds to the method that was inlined. But, in the case of advice body execution, an aspect-aware JIT compiler can prove the exact runtime type during code generation, and the guards can be eliminated for the same reason that the virtual function table need not be consulted.

All three of these optimizations are based on a combination of semantic invariants of AspectJ and specific information that, although inexpressible in byte code, is expressible in machine code and can reasonably be known inside the JIT.

4.2 Dynamic Instances and Residues

The discussion above covers the common case for advice execution, but AspectJ also supports more complicated aspect instantiation rules and pointcut semantics.

AspectJ supports the `perthis`, `pertarget`, and `percflow` clauses for aspect instantiation. In ajc, and similar byte code rewriting implementations, this difference is contained behind the static `aspectOf()`, or similar, call. In these cases, the location in memory of the aspect instance is not statically determinable, even at JIT time. Therefore, the aspect aware JIT can elect to produce essentially the same code that a byte code weaver would produce: a static call to a runtime method which will retrieve the proper aspect instance. However, it is still possible for the VM to guarantee the precise type of the aspect, and that the result of the static call is not null. So, avoiding the aspect instance null check and virtual function look-up is still possible.

Note, however, that the benefits of these optimizations are likely to be over-shadowed by the dynamic cost of the more expensive aspect look-up. It is possible to implement `perthis` and `pertarget` more efficiently by storing a reference to the aspect instance directly with the affected object in memory. This approach is similar to ajc which uses inter-type declarations to introduce a field into the object.

Dynamic residues are either automatically generated, as in the case of `cflow` conditionals, and `args`, `this`, and `target` type checks, or they are user-generated, as in the case of the `if` pointcut. In both cases, there is no additional static information during JIT compilation that provides any additional opportunities for optimization over the standard Java byte code optimizations thus these concepts are already cleanly expressible in byte code.

So, these optimizations work for common case advice dispatch, but more dynamic look-ups can overshadow the improvements. However, these checks can be optimized in other ways such as the way abc optimizes `cflow`, or using other efficient VM-based approaches such as in [5]. Further, these dispatch optimizations do not hinder the efficient implementation of the dynamic features of the language, they are in fact orthogonal, and other optimizations that perform non-local analysis can still be applied.

4.3 VM Architecture Inter-Dependencies

As mentioned at the beginning of section 4, the optimizations presented tie the compiled code to the rest of the architecture in limited and tightly controllable ways. Since the static part of each pointcut is compiled directly into native code, and the dynamic residues of the pointcut have been inlined, we have only two dependencies.

The direct dependency is on the runtime representation of the aspect; some of the optimizations above require that the aspect is instantiated and stored in a known location before the advice runs, so this optimization is only available when the runtime architecture will not move the aspect instance for the purposes of garbage collection. In practice, this is a reasonable constraint, because this optimization is performed on `issingleton` aspects only, which are created once and have a lifetime matching the lifetime of the program. Aspects must also be represented internally as annotated classes; the aspect instances must be laid out like object instances. Maintaining this layout makes advice invocation the same operation as method invocation.

The indirect dependency exists because of the ability to load aspects during the execution of the program. Since the machine code for a method can be generated before an aspect is loaded, that machine code becomes stale if the new aspect contains advice that must weave into join point shadows in the compiled method. This dependency causes a restriction on the runtime architecture to ensure that stale methods are not executed; they must be either edited or recompiled if they are stale.

This is not an issue in the current static semantics of AspectJ because all aspects must be loaded before classes are loaded into the VM, and so there will never be stale methods. However, one of the potential advantages of a runtime architecture is to support more dynamic behaviour than is specified in the AspectJ language. Stale methods would result from aspect loading in any such runtime system, because it cannot be known when a method is compiled whether it will be advised by an aspect that is not yet loaded. So, this design constraint on the architecture will be present in any runtime system which employs a JIT compiler and provides dynamic aspect loading semantics. Further, there are semantic ambiguities raised by allowing dynamic deployment that need to be resolved. One possible resolution is the `deploy` construct in Caesar [21].

```
 9 ia32_call  AF CF OF PF SF ZF = <[edi(Lorg/vmmagic/unboxed/Offset;)]+1191182812>DW,
                                    static"Aspect.aspectOf ()LAspect;"
12 ia32_mov    ebp([Ljava/lang/Object;) = <[eax(LAspect;)]+-12>DW (t13sv(GUARD))
12 ia32_add    esp(I) AF CF OF PF SF ZF <-- -4
12 ia32_call   AF CF OF PF SF ZF = <[ebp([Ljava/lang/Object;)]
                                   +[edi(Lorg/vmmagic/unboxed/Offset;)]>DW (<TRUEGUARD>),
                                  virtual"Aspect.before$0()V", eax(LAspect;)
```

Figure 1: Machine code produced for AspectJ byte code weaving for a simple before call advice

```
-11 ia32_mov  ebp([Ljava/lang/Object;) = <0+1459722656>DW (<TRUEGUARD>)
-11 ia32_mov  eax(I) = 0x570199ac
-11 ia32_add  esp(I) AF CF OF PF SF ZF <-- -4
-11 ia32_call AF CF OF PF SF ZF = <[ebp([Ljava/lang/Object;)]+60>DW (<TRUEGUARD>),
                                  virtual_exact"Aspect.before$0()V", eax(LAspect;)
```

Figure 2: Machine code produced by AJVM for a simple before call advice

5. Implementation

Our work to date is focused on the generation of efficient woven machine code. To enable this generation we have developed initial, simple implementations of the rest of the architecture (external and internal representations, memory management and dispatch). These can be thought of as simply loading pre-compiled aspects in a way similar to ajc's load-time weaving support: we load aspects stored in class files annotated with meta-data representing the additional data that is unique to aspect definitions.

We have implemented our architecture in the JikesRVM [7]. This implementation serves to validate the target model and overall architecture presented above. Our basic architecture and types of optimizations are applicable to other VM-based architectures as outlined in the previous section, although there are, of course, numerous implementation details critical to performance that are tightly coupled to the JikesRVM.

The JikesRVM has no interpreter; it uses two JIT compilers to translate Java byte code directly to native machine instructions. One of these compilers is the baseline compiler; a very fast compiler that produces code that exactly simulates the Java byte code stack machine. When the VM has determined that the cost of optimizing a method is justified, it invokes the optimizing compiler [9] to recompile that method.

Our current implementation supports before advice on both execution and call join points, in both the baseline and optimizing compiler.

In the baseline compiler, our weaver is integrated into the direct translation to native code. It checks each join point shadow against the registered set of advice and pointcuts. If a match can be determined statically, then direct calls to the appropriate advice body are inserted as dictated by the advice type (before, after, etc); any dynamic residues (such as cflow testing), are implemented as a conditional branch based on runtime information. No attempt is made to inline advice calls at this point. However, aspect instance look-up is still optimized as discussed in the previous section, by hard-coding the singleton aspect addresses into the machine code. Since the baseline compiler has little effect on frequently executed methods, our optimizations to the baseline compiler have little effect on the steady-state, long-running performance of most programs in which advised shadows execute frequently because the methods will be recompiled by the optimizing compiler.

The weaver integration into the optimizing compiler is more complicated because the optimizing compiler has several rounds of optimization and translation and is controlled by an adaptive-optimization system (AOS) [1] within the VM. The optimizing compiler uses several levels of intermediate representations. Our weaver operates during the translation from Java byte code to a high-level intermediate representation (HIR). Operating at this level gives us sufficient expressibility to address memory directly and to remove null checks and virtual function table look-ups, but, at the same time, the weaving takes place before many optimizations, such as copy-propagation and register allocation. Additionally, the inlining of advice method calls is determined by the JikesRVM adaptive-optimization system (AOS) framework. The AOS framework uses profiling data accumulated during the run of a program, in both baseline and optimized code, to drive decisions during the optimization of a method. One of the decisions affected by the profiling data is whether or not a method gets inlined.

Consider, again, the byte code produced by ajc from the previous section of a simple before advice on a call join point.

```
//Method Aspect.aspectOf:()LAspect;
9:  invokestatic    #31;

//Method Aspect.before$0()V
12:  invokevirtual   #34;
```

The JikesRVM optimizing compiler translates this code into the machine intermediate representation (MIR) shown in figure 1. MIR is the last intermediate representation before native code, and it is translated to native instructions in a straightforward manner which includes the insertion of necessary null checks and constant computation.

Looking at the code we see that the call associated with byte-code 9 retrieves the aspect instance and stores it in eax. The first instruction associated with byte code 12 moves the aspect class TiB (type information block) into `ebp`. The TiB contains the virtual method table for the class associated with the aspect. This instruction also has associated with it a guard, `t13sv`. This guard is a null check guard which is causing a null check to be output before referencing eax, which could be null.

Secondly, notice that the call instruction computes an offset from `ebp` to locate the code array to execute. This computation is the virtual function table reference.

The MIR produced by the AJVM for the same advice is shown in figure 2.

There are no associated byte code indices with this advice call. Advice calls are annotated with the special byte code index -11 to denote that they are part of the runtime architecture, and not the program being run.

Secondly, note that both `ebp` and `eax` are loaded with immediate hard-coded values. These refer to the virtual function pointer table and the aspect instance address respectively. In the call to the before advice, the pointer to the virtual function table is a constant loaded into `ebp` with constant offset (60), because we have proven that the aspect type is exact it is not necessary for us to determine this pointer dynamically. So, when this call is translated into the final native code, these constants can be computed statically into a constant address for the method.

Finally, note that all the guards in these instructions are "true guards." They generate no additional runtime checks, including the instruction that populates `eax` with a constant value. So this advice call generates no runtime null check.

From these actual generated instructions, we can see that there are no dependencies on the rest of the runtime architecture other than those mentioned in the previous section. That is, hard-coding the aspect instance address directly into the machine code depends on the fact that the aspect instance is not moved; however, there are no further dependencies on how aspects, pointcuts, and look-up tables are managed. The machine code is an exact representation of the advice and pointcut; there are no additional data structures that must be referenced, and so the representation of this information in the rest of the architecture is unconstrained.

Further, although the specific details of the implementation are specific to the JikesRVM, any VM implementation of this architecture must have these essential pieces in its implementation: virtual function look-ups, instance look-ups, type check guards, and null checks. Therefore, our approach for weaving is not specific to the JikesRVM, and it could be integrated into any VM containing a JIT compiler.

6. Results and Evaluation

This section presents the setup of our experimental benchmark suite, and evaluates the results we get from them.

6.1 Experimental Setup

We use a number of micro benchmarks to measure the performance of our target model. The micro benchmarks are configured to run until the VM reaches a steady-state. That is, we run the benchmarks a number of times without measuring performance so that the optimizing compiler has sufficient opportunity to optimize the code that we are measuring.

We start from JikesRVM version 2.4.4. The VM is built using its production build configuration which enables all JIT optimizations, compiles the entire VM with the optimizing compiler, and disables runtime assertions. The AJVM is a modified version of JikesRVM v2.4.4 built the same way. We use the timing measurement utilities from the Java Grande suite [8] of benchmarks to measure the execution time of our tests.

Our benchmarks were run on a Intel Pentium 4 3Ghz machine with 1GB of memory running SuSE Linux 10.1 with kernel version 2.6.16.

We ran benchmarks using call and execution join points on an unadvised and advised Fibonacci computation, and the cross product of the call and execution join points of a trivial method with no advice, and trivial advice with no dynamic values, and with this, target, args, and cflow dynamic values.

Figures 3 and 4 show the class, AClass, and aspect, AnAspect, used in all the benchmarks aside from the Fibonacci benchmarks. Figure 4 shows advice from all the benchmarks, but only one of these is active at any given time; they are just shown together here for brevity. The individual differences in each benchmark are explained below.

```
class AClass {

  int counter = 0;

  public void m1(int i) {
    counter++;
  }

  public int fib(int i) {
    if(i <= 1)
      return 1;
    return fib(i - 1) + fib(i - 2);
  }

  public void cflowCrossing(int numRepeats) {
    for(int i = 0; i < numRepeats; i++) {
      m1(i);
    }
  }

  public void test(int numRepeats) {
    for(int i = 0; i < numRepeats ; i ++)
      m1(i);
    //fib(6);
  }

  public void BenchRun(int numRepeats,
                       double primingTime) {
    double time = 0.0;
    JGFInstrumentor.addTimer("BCTimer",
                             "jp executions");
    while(time < primingTime) {
      JGFInstrumentor.resetTimer("BCTimer");
      JGFInstrumentor.startTimer("BCTimer");
      test(numRepeats);
      JGFInstrumentor.stopTimer("BCTimer");
      time += JGFInstrumentor.readTimer("BCTimer");
      JGFInstrumentor.addOpsToTimer("BCTimer",
                                    (double)numRepeats);
    }
    JGFInstrumentor.printTimer("BCTimer");
  }
  public static void main(String [] argv) {
    int numRepeats = 10000000;
    int primeForS = 10;
    if(argv.length > 1) {
      numRepeats = Integer.parseInt(argv[0]);
      primeForS = Integer.parseInt(argv[1]);
    }
    (new AClass()).BenchRun(numRepeats, primeForS);
  }

}
```

Figure 3: The Class and Driver used in the micro benchmarks

The graphs in figure 5 show the results of all the benchmarks. Each graph measures the time in seconds that it took each implementation to finish executing the benchmark. All times are normalized to ajc compiler implementation.

The following abbreviations are used in the graph: TM for Trivial Method, TA for Trivial Advice, and Fib for Fibonacci.

The Trivial Method benchmark is a control benchmark to calibrate normal virtual method calls in the JikesRVM, on our platform. In this case, AnAspect is not loaded at all, and `AClass.m1()`, a method which increments a counter, is run repeatedly in a loop. This benchmark is used to ensure that all implementations perform equally well when there are no applicable advice in the system.

The second benchmark defines a trivial advice, which also increments a counter on the call or execution join point of the trivial

```
public aspect AnAspect {

  private int counter = 0;

  /* Trivial Advice (TA).
   * Used in TMTA and FibTA benchmarks
   */
  before ():
  call/execution(void AClass.m1/fib(..)) {
    counter++;
  }

  /* Trivial Value Advice (TAwVAL).
   * Used in this, target, args.
   */
  before (int x):
  args (x)
    && call/execution(void AClass.m1 (..)) {
    counter += x;
  }

  before(AClass obj):
  this/target (obj)
    && call/execution(void AClass.m1 (..)) {
    counter = obj.counter ;
  }

  /* Trivial Advice Dynamic residue dispatch (TADyn)
   * Used in perthis, inside cflow, outside cflow
   */
  before () :
  cflow (call/execution(void AClass.cflowCrossing()))
    && call/execution(void AClass.m1(..)) {
    counter++;
  }
}
```

Figure 4: The Aspect used in the micro benchmarks

method, and the time measurement of the loop is retaken. In this case, the Aspect is deployed in the system.

The Fibonacci benchmark measures the execution time of computing Fibonacci numbers in a loop without any advice. In this case, we call `AClass.fib()`, and AnAspect is not deployed in the system. This benchmark ensures that each implementation performs equally well where there are no aspects present. The second Fibonacci benchmarks introduces the aspect to the system and retakes the measurement.

Both the Trivial Method and Fibonacci use the Trivial Advice defined in the aspect in figure 4.

Micro-measuring both the trivial method and the Fibonacci advised methods allows the numbers to reflect different possible optimizations. Specifically, Fibonacci is not amenable to inlining because it is a recursive function.[3] On the other hand, calls to the trivial method are straight forward to inline, and we can reasonably expect a JIT to do so. Measuring both of these cases ensures we get an accurate reflection of the cost of the advice dispatch, not how well the optimizing compiler inlines method calls.

The second section of the benchmarks compares the efficiency of the different implementations by measuring the time it takes to execute join points where dynamic information is needed as part of advice dispatch. We show measurements that pick out the `this` object, the `target` object, and the `args` of the method call. In these cases, one of the advice from the Trivial Value Advice in figure 4 is used.

[3] It is amenable to loop-unrolling style optimization, however our benchmark runs it at a depth (6) that we believe is sufficient enough to discourage unrolling the entire recursion.

The last section of benchmarks shows two more complicated constructs in AspectJ: `perthis` aspect instantiation, and the `cflow` pointcut. The `perthis` benchmark is the same as the trivial method and trivial advice benchmark mentioned above except that the aspect is instantiated by the `perthis` instantiation rules.

In benchmarking the `cflow` pointcut we used two different benchmarks. The `cflow` pointcut is shown in the last section of figure 4 and is used in both cflow benchmarks, inside cflow and outside cflow. These tests repeatedly call the `AClass.m1()` method and use the same advice as in the previous cflow benchmark. In the first case, these calls are done within the control flow of the `cflowCrossing`, and in the second case they are not. These two benchmarks measure the cost of any sort of runtime checking required to determine the control flow context. Note, however, that this runtime checking is not specifically part of the advice dispatch, but it does demonstrate that our implementation does not hamper the efficient implementation of dynamic residues.

The full numbers from each benchmark have been included in the appendix A in table 1.

6.2 Evaluation

First we notice that there is little variation in the running time of all 4 of the benchmarks which are executed with no advice in the system: Trivial Method and Fibonacci for both call and execution join points. This fact is important because it demonstrates that integrating advice dispatch into the JIT does not slow down unadvised method dispatch.

Secondly, also in the Trivial Advice benchmarks, we see that the AJVM performs consistently better in all four cases that include advice in the program: both Trivial Method plus Trivial Advice and Fibonacci plus Trivial Advice. These results show that advice dispatch overhead is recovered whether or not the JIT can inline the advised virtual method call. Looking at Table 1 of absolute execution time in appendix A, we can see that the improvements range from 6% to 8.6% for Trivial Method plus Trivial Advice and 2.3% to 5.1% for the Fibonacci plus Trivial Advice.

The fact that the By Hand implementation does better than AJVM in the Trivial Method plus Trivial Advice (TM&TA) call and execution benchmarks is due to differences in the way the optimizing compiler unrolls the loops in `AClass.m1(..)`. The loops are unrolled differently because the code the compiler operates on is different between the two implementations, and hence different decisions are made. Most notably, AJVM does not output null checks for the aspect instance, but the By Hand implementation does.

However, our optimizations do improve on the advice dispatch for this benchmark demonstrated by Figure 6. This graph shows the running time of Trivial Method plus Trivial Advice for just the AJVM and By Hand implementations with the optimizing compiler turned off. We can see that our optimizations improve on the By Hand implementation when no other optimizations are applied. Therefore we can expect that integrating knowledge of advice dispatch into the dynamic optimizing compiler could further improve its performance by directing loop unrolling and other optimizations.

The second row of graphs Figure 5 show the costs of executing Trivial Advice that use a dynamic value as one of their arguments. These results show slightly better improvements of the AJVM over ajc than in the Trivial Advice cases for the `this` and `target` pointcuts. We do see more significant improvements in the `args` pointcut. This gain can be also be attributed to different loop unrolling strategies used by the optimizing compiler. In this case, the differences between the aspect instance look-up implementations caused the optimizing compiler to choose a more efficient loop unrolling layout for the AJVM.

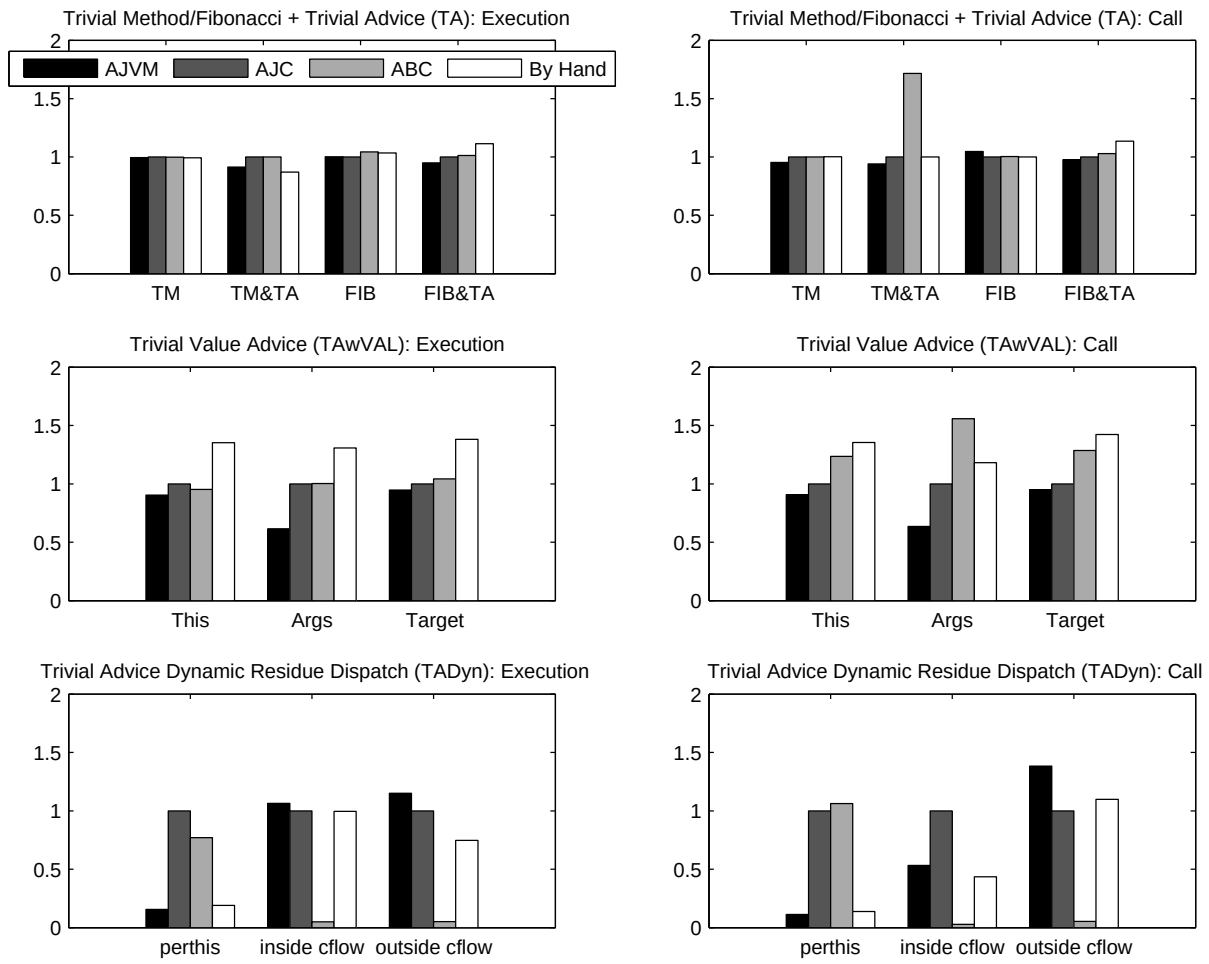

Figure 5: Execution time in seconds normalized to ajc TM=Trivial Method; TA = Trivial Advice; FIB = Fibonacci

The perthis benchmark shows that we can achieve much greater performance when the VM understands how aspect instances should be managed. The AJVM uses a private field in affected objects to store the aspect instance associated with that object in the case of `perthis` aspect instantiation. In this case the aspect-aware JIT outputs a simple instruction to retrieve directly the instance pointer that is stored with the object. A null check determines if the aspect has already been created. However, ajc uses inter-type declarations and an interface on the object to manage the aspect instance associated with the pointcut. This management adds considerable overhead that is avoided in the machine code model.

While looking at the cflow benchmarks, it is important to realise the difference between the strategy of the cflow implementation, and the actual advice dispatch. The AJVM implements optimizations that improve the advice dispatch, and in the case of cflow, it employs the same strategy (thread local counters) as ajc to determine at runtime whether the advice body needs to be executed. In this case, the cost of maintaining the counters overshadows the advice dispatch, so that the AJVM and ajc come out with similar, but varying results. This variability stems from the subtle interaction between the dynamic optimizing compiler and the cflow implementation strategy, rather than the cost of advice dispatch.

Furthermore, we see that abc's non-local strategy for cflow optimization has a very significant increase in the performance of the micro benchmarks. This optimization can prove, in some cases, that no thread local counter (or stack) need to be consulted or managed at all, and hence we can remove all runtime checks guarding the advice dispatch. There is no reason why the strategy used in [5] could not be integrated into AJVM to replace its naive thread local counter strategy. However, in this paper we focus on recovering overhead caused by the mismatch between AspectJ semantics and Java byte code specifically.

The cflow benchmarks, do, however demonstrate that the AJVM is comparable to the ajc implementation. Therefore, we can conclude that our advice dispatch optimizations do not preclude further orthogonal optimizations related to the dynamic features of the language.

7. Related Work

Object-Oriented runtime architectures are well studied. This work is discussed in detail in section 3, and so we will not repeat it here.

Runtime support for aspects is a natural progression as AOP becomes more widespread and adopted; AspectJ would not have been adopted if the first implementation involved a customized VM.

The first implementations of AspectJ were based on preprocessing; this was followed by byte code rewriting based ap-

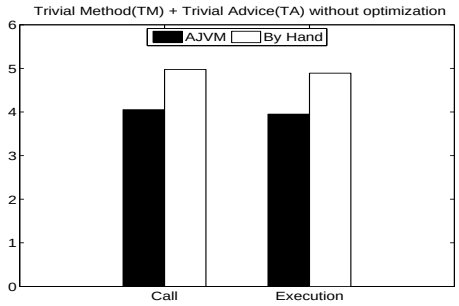

Figure 6: Trivial Method plus Trivial Advice execution time in seconds without dynamic recompilation and optimization

proaches [19, 22]. All of these sorts of implementations suffer from the problem that the output semantics are constrained to a language that is too high level to be optimally efficient.

Byte code rewriting implementations typically compile aspects into classes, advice bodies into methods, and handle advice execution by inserting appropriate dynamic checks and method calls into the program being rewritten. However, byte code rewriting implementations suffer from a similar problem to pre-processing techniques because the Java byte code semantics do not include instructions that can express cleanly AspectJ semantics.

AspectWerkz [6] implements an AspectJ-like language on top of traditional Java using a separate XML file, or embedded annotations to specify advice and pointcuts. AspectWerkz, like ajc, can use a static or load time weaving approach together with a runtime library. However, AspectWerkz makes heavy use of delegation and reflection on advised join points which make it simple to make changes to the aspect and advice (not, however, pointcuts) at runtime. This method of weaving introduces extra indirection costs associated with advice execution that significantly hamper its performance. Our work instead focuses on efficient advice dispatch which can be used to support an efficient dynamic weaving system like AspectWerkz.

Prose [24], unlike ajc or AspectWerkz which use a modified class loader together with a runtime framework on top of standard Java VMs, adds hooks into the VM so that an AOP system could be implemented efficiently on top of it. Prose works by instrumenting the JikesRVM baseline JIT compiler to weave stubs into each join point shadow as the code is compiled. Each stub calls out to the AOP engine which can arrange for advice to be executed or not as required. This approach differs from ours in that we instrument only the join point shadows which can have advice associated with them, and further, we do not weave stub method calls to these join points, but rather inline the code to execute the advice body or the dynamic residue required for any runtime checking.

Steamloom [4, 15] moves advice weaving into the virtual machine. However, it still implements advice dispatch using byte code rewriting. The advantage that Steamloom gains by doing the byte code rewriting in the virtual machine is the added dynamism available because the virtual machine understands what aspects and advice are. This allows Steamloom to achieve greater cflow performance in some cases. However, the limits of byte code rewriting still constrain Steamloom. As described in [14], Steamloom seeks to partially address this by adding a new internally used byte code to the JIT to improve the performance of aspect instance look-up.

AspectWerkz, Prose, and Steamloom are approaches for dynamic weaving AOP systems. The work we present in this paper continues on the path created by these implementations by moving advice one step further into the JIT compiler.

8. Future Work

Achieving efficient advice dispatch in a runtime environment raises many questions regarding the architecture and implementation of the rest of the runtime environment supporting AspectJ: the external and internal representations for aspects, the dispatch and weaver, and memory management. These questions parallel the basic design of VM-based OO architectures, and lead to two basic avenues of future work: the efficient design and implementation of the supporting architecture, and revising the semantics of AspectJ to accommodate the inherit dynamic properties enabled by this architecture.

The internal representation of aspects affect all other parts of the architecture, and it determines the primary trade-offs for space and time efficiency when looking up aspect-related data. The algorithms and representations used define how quickly pointcuts can be matched or searched, and these operations are critical to support the efficient operation of the dispatcher and JIT compiler. This overhead can be a primary concern in short lived programs that require low start-up times in which there is less time for pointcut related computations to be amortized over the life of the program, and for programs that dynamically load and unload aspects during the life of the program.

The internal representation of aspects further drives the design of the external representation so that aspects can be loaded quickly from the external format. It is conceivable that a static AspectJ compiler could pre-process and layout advice and pointcuts to minimize the work that needs to be done during aspect load time for conversion to the internal representation, and for determining which previously loaded methods must be modified because of the new advice.

Further, as mentioned in section 3, we have implemented an aspect-aware JIT compiler, but not an aspect-aware adaptive-optimization system. There could still be performance gains by integrating advice dispatch into the AOS. These gains are especially likely when optimizing dynamic residues from runtime type matching for `args`, `target` and `this` pointcuts, and cflow checking. Integrating advice dispatch further into the AOS moves aspect-aware runtime environments further along the path followed historically by OO runtime environments.

As further validation, we intend to work with a commercial JVM vendor to replicate our experimental results on a production quality JVM.

Additionally, a proper runtime architecture potentially enables more dynamic language semantics. This dynamism can include the ability to load and unload aspects, control the activation of advice, and possibly allow the ability to define pointcuts and advice at runtime. But, AspectJ's semantics do not wholly apply to the new environment, and several important semantic questions need to be addressed. These include dealing with concurrency issues and advice activation. In these cases, there are subtle issues that arise with atomicity of advice deployment, and how the deployment interacts with threads that have newly advised live join points on their stacks. Many of these questions have been addressed in Caesar [21] using their `deploy` construct.

There is work to be done in determining whether inter-type declarations, or other join point models, should be supported by a runtime environment, or if they should only be a feature of a static byte code compiler. These problems directly affect AspectJ semantics, and they will need to be addressed to maintain consistent language semantics when deployed in a runtime architecture.

Finally, our current implementation supports only before advice on call and execution join points. The remaining join points are not very conceptually different than call and execution. However, advice types like around are significantly different and could benefit significantly from integration into the JIT.

9. Conclusion

We have discussed the semantic mismatches between AspectJ and Java byte code. These mismatches stem from the inability to express the exact type of a runtime object, or to provide guarantees that an object exists to avoid null checks at runtime in Java byte code. Although the inability to express these concepts does not restrict the byte code rewriting based implementations of AspectJ functionally, we have shown that they do impose performance constraints.

We have presented a machine code model that can be targeted by virtual machine JIT compilers that alleviates these inefficiencies. We verified our claim that this model can be targeted by a JIT compiler by providing an implementation based on the JikesRVM in which we modify its baseline and optimizing JIT compilers to target our machine code model. Further, we have shown that both the target machine code model, and our implementation, rely on only two reasonable constraints on the supporting runtime architecture; this fact allows us to pursue future work on more fully integrating AspectJ into a runtime environment.

Finally, we have verified that these performance constraints are alleviated by our target model by showing the results of a suite of micro benchmarks which compare the running times of advice dispatch in different scenarios across different weaving implementations.

Our work progresses along the same path that OO runtime architectures progressed in the past 25 years. We hope that by using OO architecture work as a guide, we can quickly bring VM level support for languages like AspectJ to a similar state.

Acknowledgments

This research was partially supported by the IBM Centers for Advanced Studies and the Natural Sciences and Engineering Research Council (NSERC).

A. Results

Table 1 shows the absolute numbers produced from the benchmarks discussed in section 6. The results show the absolute running time in seconds of each of the benchmarks on each of the implementations.

References

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–65, New York, NY, USA, 2000. ACM Press.

[2] Matthew Arnold, Adam Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 297–311, New York, NY, USA, 2005. ACM Press.

[3] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[5] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient control flow quantification. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 125–138, New York, NY, USA, 2006. ACM Press.

[6] Jonas Bonér and Alexandre Vasseur. AspectWerkz. `http://aspectwerkz.codehaus.org/index.html`.

[7] Bowen Alpern and C. R. Attanasio and Anthony Cocchi and Derek Lieber and Stephen Smith and Ton Ngo and John J. Barton and Susan Flynn Hummel and Janice C. Sheperd and Mark Mergen. Implementing jalapeo in Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 314–324, New York, NY, USA, 1999. ACM Press.

[8] Bull. A benchmark suite for high performance Java. *Concurrency, practice and experience*, 12(6):375, 2000.

[9] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM Press.

[10] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM Press.

[11] Frequently Asked Questions about AspectJ. `http://www.eclipse.org/aspectj/doc/released/faq.html`, 2006.

[12] Adele Goldberg and David Robson. *Smalltalk-80: Ihe Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[13] James Gosling, Gilad Bracha, Bill Joy, and Guy L Steele. *The Java Language Specification*. Addison-Wesley Professional, 2000.

[14] M. Haupt and M. Mezini. Virtual Machine Support for Aspects with Advice Instance Tables. *L'Objet*, 11(3):9–30, 2005.

[15] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An Execution Layer for Aspect-Oriented Programming Languages. In J. Vitek, editor, *Proceedings of the First International Conference on Virtual Execution Environments (VEE'05)*, pages 142–152, Chicago, USA, June 2005. ACM Press.

[16] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[17] Urs Hölzle and David Ungar. A third-generation self implementation: reconciling responsiveness with performance. *SIGPLAN Not.*, 29(10):229–243, 1994.

[18] Gregor Kiczales and Erik Hilsdale. Aspect-oriented Programming. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, page 313, New York, NY, USA, 2001. ACM Press.

[19] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[20] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specificaiton*. Addison-Wesley Longman, Inc., 1997.

[21] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[22] Pavel Avgustinov and Aske Simon Christensen and Laurie Hendren and Sascha Kuzins and Jennifer Lhoták and Ondřej Lhoták and Oege de Moor and Damien Sereni and Ganesh Sittampalam and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented*

	AJVM	AJC	ABC	BH
execution join points				
Trivial Method	2.181	2.191	2.187	2.176
Trivial Method and Trivial Advice	3.562	3.899	3.902	3.391
Fibonacci	28.751	28.707	29.945	29.681
Fibonacci and Trivial Advice	29.975	31.575	31.976	35.151
this(x)	11.45	12.683	12.082	17.149
args(x)	11.237	18.28	18.321	23.903
target(x)	11.547	12.185	12.706	16.834
perthis	6.981	44.761	34.51	8.542
inside cflow	23.279	21.883	1.08	21.769
outside cflow	22.876	19.879	1.031	14.844
call join points				
Trivial Method	2.093	2.195	2.195	2.198
Trivial Method and Trivial Advice	3.534	3.761	6.457	3.761
Fibonacci	29.743	28.433	28.534	28.452
Fibonacci and Trivial Advice	30.403	31.133	32.017	35.356
this(x)	11.521	12.703	15.688	17.203
args(x)	11.52	18.116	28.211	21.406
target(x)	11.423	12.024	15.467	17.108
perthis	6.989	62.284	66.183	8.58
inside cflow	23.347	43.801	1.222	19.062
outside cflow	22.906	16.572	0.877	18.191

Table 1: Running time of micro benchmark tests in seconds

software development, pages 87–98, New York, NY, USA, 2005. ACM Press.

[23] Pavel Avgustinov and Aske Simon Christensen and Laurie Hendren and Sascha Kuzins and Jennifer Lhoták and Ondřej Lhoták and Oege de Moor and Damien Sereni and Ganesh Sittampalam and Julian Tibble. Optimising AspectJ. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[24] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for Java. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 100–109, New York, NY, USA, 2003. ACM Press.