Interpreter Implementation of Advice Weaving

UBC Technical Report TR-2010-01

Immad Naseer
University of British Columbia
inaseer@cs.ubc.ca

Ryan M. Golbeck
University of British Columbia
rmgolbec@cs.ubc.ca

Peter Selby
University of British Columbia
pselby@cs.ubc.ca

Gregor Kiczales
University of British Columbia
gregor@cs.ubc.ca

ABSTRACT

When late-binding of advice is used for incremental development or configuration, implementing advice weaving using code rewriting external to the VM can cause performance problems during application startup.

We present an interpreter-based (non-rewriting) weaver that uses a simple table and cache structure for matching pointcuts against dynamic join points together with a simple mechanism for calling the matched advice.

An implementation of our approach in the Jikes RVM shows its feasibility. Internal micro-benchmarks show dynamic join point execution overhead of approximately 28% in the common case where no advice is applicable and that start-up performance is improved over VM-external weavers. The cache and table structures could be used during later (i.e. JIT time) per-method rewrite based weaving to reduce pointcut matching overhead. We conclude that it is worthwhile to develop and evaluate a complete in-VM hybrid implementation, comprising both non-rewriting and rewriting based advice weaving.

Categories and Subject Descriptors

D.3.4 [**Programming languages**]: Processors—*interpreters, run-time environments, optimization*

General Terms

Performance, Languages

1. INTRODUCTION

Pointcut and advice functionality is an important part of several aspect-oriented programming (AOP) [19] languages and toolkits, including AspectJ [18], CaesarJ [23], JBoss AOP [17], and Spring [25]. The implementation of pointcut and advice semantics requires *weaving* advice execution into well defined points in the program's execution. These points of execution are called *dynamic join points* (DJPs). Prior work on AspectJ and other AOP frameworks has implemented advice weaving at compile time using source or bytecode rewriting [15, 2]; at load time using bytecode rewriting [18, 11]; during runtime using bytecode rewriting, including the ability to do dynamic deployment of aspects with advice [4]; and during just-in-time (JIT) compilation by augmenting the machine code generation process [12]. Work on Spring, JBoss, JAsCo, JRockit, PROSE, and Nu [25, 17, 26, 28, 24, 8] has implemented advice weaving using runtime lookup and invocation.[1]

Each dynamic join point in a program's execution has a corresponding *DJP shadow* in the program text [15]. In rewriting approaches, DJP shadows are augmented with additional code to execute advice bodies appropriately. Code rewriting prior to execution effectively compiles out the static overhead of advice lookup, offering the advantage of better runtime performance, at the cost of requiring a global scan of the code to find potentially advised shadows and rewriting code at such shadows. This can cause problems for interactive development—if only part of the program runs, the overhead of scanning and potentially rewriting all the code may outweigh the savings of more efficient advice execution.

Incremental weaving is intended to better support interactive development, but it does not address late aspect deployment. When the aspect configuration is not known until system startup, rewriting is done at load time (load time weaving). This approach must scan and potentially rewrite all application code loaded by the VM, whether it is executed or not, impairing startup performance.

Runtime bytecode rewriting is another option. However, many VMs, including IBM's J9 VM, store loaded bytecode in a read-only format, which renders this method impractical. Because such VMs are widely used, we are interested in exploring alternative methods for implementing advice weaving that can be used even when bytecode rewriting is not possible.

VMs which store bytecode in a read-only format often implement an interpreter to improve startup performance. We call these *hybrid* virtual machines, since they combine an interpreter with an optimizing JIT compiler, allowing them to combine good startup performance and good long-term steady state performance. The existence of an interpreter in these VMs suggests that an interpreter-based approach to advice weaving might be simpler and more effective.

The work presented here is part of a project exploring whether a *hybrid implementation strategy* for advice weaving can pro-

[1]We use the term weaving to encompass all approaches to coordinating advice execution with DJPs, including rewriting based and runtime lookup and invocation based approaches.

	call		execution		get, set		other		*Overall*	
AJHotDraw	< 1	11	1	2	–	–	–	–	< 1	5
aspectJEdit	3	< 1	3	< 1	< 1	–	1	< 1	2	< 1
observer-project	2	4	–	–	–	–	–	–	< 1	2
telecom-project	5	6	–	–	–	–	–	–	2	2
tracing-project	–	–	32	4	–	–	–	–	6	< 1
abc benchmarks	17.25	13.46	16.18	< 1	0.64	< 1	–	–	9.30	4.49
Overall	4	13	5	< 1	< 1	< 1	< 1	< 1	3	4

Table 1: Frequency of advice applicability at shadows and DJPs. The first two rows are application programs, the next three are from the AspectJ examples and the last are the combined total of 14 from the abc benchmark suite[2]. For each kind of DJP, the left sub-column is the percentage of advised shadows and the right sub-column is the percentage of advised DJPs. The *other* column includes initializer, static initializer, preinitializer, handler and advice execution DJPs. The *Overall* row and column show the percentages of advised shadows and advised DJPs taken over all input programs and all DJP kinds respectively.

vide a good balance between quick start up and optimal steady state execution efficiency in these types of VMs. AspectJ implementations could use an interpreter-based strategy for advice weaving for quick startup, and then transition to a rewriting-based approach as part of the JIT compiler, thereby integrating advice weaving with the existing runtime compilation architecture.

This paper addresses one key part of the overall question: how to architect an efficient interpreter implementation of pointcut and advice weaving. By *interpreter implementation* we mean: (i) little or no work is required from the static compiler to implement weaving; (ii) little additional work is required from the loader; (iii) existing bytecode interpreters can be straightforwardly extended to work this way; and (iv) none of the static compiler, loader or interpreter must perform a global scan typical of rewriting approaches.

Furthermore, although the performance of the interpreter itself is important, it is not essential to the overall performance of the hybrid VM because long running code will be optimized by the JIT compiler. Rather, the main goal of an interpreter is to startup quickly, have low memory overhead, and to assist the JIT compiler with profiling information [22]. We observe these goals in our design.

The contributions of this paper are: a simple analysis of advice frequency in AspectJ programs; a design for interpreter advice weaving; an implementation in the Jikes RVM; and an analysis that suggests the performance is good enough to warrant development of a complete hybrid implementation.

2. ADVICE APPLICABILITY IN ASPECTJ PROGRAMS

The relative frequency of advice being applicable at DJPs is an important factor in our design. Table 1 shows advice applicability information gathered from two applications (AJHotDraw [20] and aspectJEdit, a version of jEdit refactored using AspectJ [16]); three of the AspectJ example programs (telecom, observer and tracing); and 14 benchmarks of the abc benchmark suite [13].[2] Of course the benchmarks are not actual applications, and so they reflect idiosyncratic behaviour typical of benchmarks. But we include them for completeness, and we note that collectively they support the same conclusions as the other programs.

The data was gathered by instrumenting the ajc load time weaver to collect the static data and making the load time weaver itself to instrument the application code to collect the dynamic data. In the table, we include counts for regular AspectJ advice, special counter manipulating advice generated to handle cflow pointcuts, as well as other special advice generated by ajc to handle different modes of advice instantiation.

The data shows that most DJPs and shadows have no matching advice. This motivates a *fast fail strategy* in our design, by which we mean that the design is biased towards being able to determine quickly that no advice applies at a shadow and then proceed normally. The collected data also shows that over all the applications and benchmarks in the table, call and execution DJPs comprise 32% and 22% of total DJPs respectively; and call and execution shadows comprise 50% and 12% of total shadows.[3] This is not shown in Table 1.

3. WEAVING ARCHITECTURE

Our architecture comprises several elements which we describe in the order they arise as aspects are loaded and the application starts to run. In the discussion we use the following terminology. Pointcuts *match* against DJPs by testing attributes of the pointcut against properties of the DJP. Different kinds of DJPs have different properties, which can include modifiers, the static type of the target (STT), the target name (TN) and others. All kinds of DJPs except advice execution have a STT and 4 out of 11 kinds of DJPs have a TN. In the pointcut `call(void C*.m*())`, the `C*` is a type pattern that matches against the STT, or an *STT type pattern* for short. Similarly, the `m*` is a *TN name pattern.*

3.1 STT Type Pattern Tables

[2]The 14 are: ants, cona-sim, cona-stack, dcm-sim, nullcheck-sim, nullcheck-sim_after, nullcheck-sim_notwithin, nullcheck-sim_orig, quicksort_gregor, quicksort_oege, bean, bean_gregor, figure and hello. The remaining abc benchmarks made no use of aspects, or could not be compiled using ajc (version 1.5.4) and so were not included.

[3]The fact that there are more calls and executions than gets and sets appears to come from the Java coding style of defining getter and setter methods which means most gets and sets have an associated call and execution.

```
class C {              class Sup {                class Sub extends Sup {
  void m()  {..}         void m()  {..}             void mSub() { .. }
}                        void mSup() { .. }       }
                       }

        --- advice ---                                   --- triples ---

(1)  before(): call(void C.m())    { ... }               <{call} {C}      {m}>
(2)  after (): call(void Sup.mSup())    { ... }          <{call} {Sup}    {mSup}>
(3)  before(): call(void Sup.* ())
               && call(void *.mSup()) { ... }            <{call} {Sup}    {mSup}>
(4)  before(): call(void Su*.mSu*))                      <{call} {Su*}    {mSu*}>
(5)  before(): call(void *.*    ())  { ... }             <{call} {*}      {*}>
(6)  before(): execution(void *.mSu*())  { ... }         <{exec} {*}      {mSu*}>

    -- STT Type Pattern Tables (TPTs) --                 -- *.tnp List --

   C:  [ 1      |     |   |   ]                            (6)
 Sup:  [ 2, 3   | 6   |   |   ]
 Su*:  [ 4      |     |   |   ]
   *:  [ 5      |     |   |   ]
 Sub:  [        | 6   |   |   ]

 -- Type Pattern Table Lists (TPTLs) --

   C:  ( C:[1  |  |  |  ], *:[5  |  |  |  ] )
 Sup:  ( Sup:[2, 3  | 6 |  |  ], Su*:[ 4  |  |  |  ], *:[5  |  |  |  ] )
 Sub:  ( Sup:[2, 3  | 6 |  |  ], Su*:[ 4  |  |  |  ], *:[5  |  |  |  ], Sub:[  | 6 |  |  ] )
```

Figure 1: A simple example and the corresponding lookup table structures. Note that the vertical bars divide the TPTs into four parts used to group advice by the kinds of DJPs they can match. Also note that advice 6 is added to the Sup and Sub TPTs during TPTL construction. The Sub TPT is created when the Sub TPTL is built.

AspectJ does not define a dynamic deployment semantics for advice. While there have been proposals for this [23, 17], we have chosen to focus on implementing the AspectJ semantics. As such, we assume the existence of aspect configuration files that specify which aspects should be loaded. At application startup, these aspects, with the advice they contain, are loaded first, before execution of the application's entry point method begins.

As shown in Figure 1, when each advice declaration is loaded, the pointcut is analyzed to produce a conservative upper bound of the DJP kinds, the STT type patterns and the TN name patterns the pointcut can match. This is done by a simple abstract evaluation of the pointcut which produces a triple consisting of three sets: (i) the kinds of DJP, (ii) the STT type patterns and (iii) the TN name patterns. For a primitive pointcut like `call(void C1.m1())` the abstract evaluation produces `<{call}, {C1}, {m1}>`. Pointcuts that do not match on the DJP kind, STT or TN (such as `within`, `cflow`, `target` etc.) return a triple of the form: `<{*},{*},{*}>`. The logical combinators `&&`, `||` and `!` produce a conservative bound over each value in the triples. Complex intersection of patterns is not performed, and instead we estimate intersection with union. This bound could be improved upon by a more sophisticated algorithm for intersecting the patterns across the sets. As part of the abstract evaluation user defined named pointcuts are expanded inline.

Once the advice pointcut has been abstractly evaluated the advice declaration is added to a set of *STT type pattern tables* (TPTs) and possibly a special **.tnp list* as follows. The advice declaration is added to the TPT for every non-wildcard STT type pattern that appears in the STT set. Advice declarations that have a wildcard (`*`) in the STT set but have one or more non-wildcards in the TN set are added to the *.tnp list. Advice declarations with a wildcard in both the STT and TN set are added to the TPT for the `*` type pattern. If the TPT for a type pattern does not yet exist it is created. The TPTs themselves are divided into four parts, for advice that can match only call, only execution, a kind other than call or execution, or any kind of DJP respectively.

3.2 TPT Lists

After the advice have been loaded normal execution of the program can begin. All the remaining lookup data structures are created lazily as they are needed. The next data structure we describe, which depends directly on TPTs and the *.tnp list is the per-type *(STT) TPT list* (TPTL). Unlike the TPTs, by the time the TPTL for a type T is created, T will be fully resolved.

To construct the TPTL for T, first the *.tnp list is scanned, looking for advice for which the target name list (third element of the triple) includes a target name defined in T. Any such advice are added to the TPT for T, which is created if it did not already exist. For example as shown in Figure 1 advice 6 is added to the TPTs for `Sup` and `Sub` when each corresponding TPTL is computed.

Then the complete list of TPTs is scanned to match the table's type pattern against T. Any matching TPTs are added to the TPTL. Note that this matching includes subtyping. For example as shown in Figure 1 the TPTL for `Sub` includes the TPT for `Sup`.

The net effect of this two part process is that the the TPTL for a type T contains a reference to every TPT that includes advice which might match shadows at which T is the static type of the target – thus all potentially applicable advice is reachable through the TPTL. The effect of the *.tnp list is to construct implicit types for `*.name` patterns, and thereby reduce the spread of a pointcut such as in advice 6 in Figure 1 to the same as the pointcut in advice 4.

3.3 Fast Fail Path

Up to this point, Section 3 has described the table structures created to lookup advice at DJPs. We now describe how these structures are used at DJP execution time.

Every DJP shadow kind corresponds to a well defined set of locations in the bytecode [15]. We extend the interpreter code that executes each kind of shadow with a small amount of code to implement the *fast fail path* of the architecture. If this code can quickly prove that no advice applies at the shadow, it will proceed to execute the normal VM instructions for the shadow. Failing that, it will take the out of line (slow) path.

The first test performed by the fast fail path code is to check the TPTL of the STT. This is straightforward because all instructions corresponding to shadows have an STT. From the STT, we fetch the TPTL, which we store as an additional field of the type. If the TPTL is empty no advice can apply; this is the fastest fail case. If the TPTL is uninitialized control goes to the out of line (slow path) code to build the TPTL and continue with a slow lookup and possible invocation.

If the TPTL is non-empty, the fast fail path then checks the per-method cache. If the cache has not been initialized, or the cache entry for the shadow has not been initialized, or the cache entry is not empty, control goes to the out of line slow path. If the cache entry is empty, it proceeds to execute the VM instructions corresponding to the shadow normally; this is the second fast-fail case.

3.4 Out of Line (Slow) Lookup

The out of line lookup path is called in any case in which the fast fail path could not be followed. For simplicity, the same lookup routine is called in each condition in which the fast fail path fails; the time savings of having multiple entry points do not appear to warrant the additional code required in the fast fail code. This means that the first step performed by the out of line path is to duplicate the fast fail path to determine which condition led to the out of line call.

If the TPTL for the STT has not been constructed, the out of line path constructs it. If the per-method cache has not been initialized or the cache entry for this shadow has not been initialized, it goes through each TPT in the TPTL of the STT, collecting matching advice from each TPT. It uses the kind of the DJP shadow to limit its search to only the relevant part of the TPTs.

If any advice is found to be statically applicable at the shadow, a *dispatcher* is constructed for each applicable advice. Any residual dynamic tests are moved into the dispatcher where they guard the invocation of the corresponding advice. This includes dynamic tests for `args` as well as those for `target` and `this` which could not be statically evaluated given the DJP shadow; it also includes the testing of cflow counters and boolean expressions specified in if pointcuts. The dispatcher handles passing of dynamic values to advice methods by getting those values off the stack. The exact locations of the dynamic values for the different DJP shadow kinds are as specified in [15].

The slow path bundles the dispatchers for all applicable advice into a *dispatcher caller* responsible for calling the dispatchers around, before and after execution of the DJP. If the dispatcher caller has to call any advice around or after the DJP it executes the DJP from within its dynamic extent and skips execution of the original DJP when returning from the slow path.

The dispatcher caller is stored in the cache entry for the DJP shadow. The cache line is left empty if no advice statically matched the DJP shadow.

If the cache entry is non-empty, the slow path calls the dispatcher caller which in turns calls the applicable dispatchers around, after and before the DJP.

4. IMPLEMENTATION

To evaluate our interpreter weaving architecture we implemented it using version 2.9.2 of the Jikes Research Virtual Machine (RVM) [7]. Our implementation supports all five kinds of advice and all DJPs except initialization, preinitialization, static initialization and advice execution. It presently supports only the singleton advice instantiation model, and does not support `thisJoinPoint`. Intertype declarations are outside the scope of this work.

We are interested in exploring weaving in hybrid VMs which store bytecode in a read-only format and implement an interpreter. The Jikes RVM does not implement an interpreter – it uses a simple *baseline compiler* in place of an interpreter. However, this compiler uses a simple template based compilation strategy to translate Java bytecode to machine code. The generated code thus looks like the unrolling of a true interpreter dispatch loop, in which the instruction dispatch overhead has been compiled out by copying machine code templates, but some values which a true interpreter would fetch from the bytecode are copied into the generated ma-

[3]In AspectJ, the syntax of a type pattern *tp* is more complex than suggested here. It can include compound type patterns which have to be processed further to get the list of STT type patterns.

chine code because the machine code does not have access to the bytecode when it runs. We can therefore consider ourselves to be developing a reasonable simulation of an interpreter in the baseline compiler as long as we follow the same coding conventions.

The code generated by the baseline compiler has essentially the same performance as an interpreter in which the main dispatch loop was written in machine code would have, except that the instruction dispatch overhead has been compiled out[4]. In addition, there is a significant code bloat factor as a new copy of the machine code is generated for every occurrence of a bytecode in any baseline compiled method.

We modified the baseline compiler to generate extra instructions for weaving advice at DJP shadows. Similar instructions would be written for an interpreter written in machine code as part of the dispatch loop. The generated machine code implements the fast fail path and calls the out of line path if required. It performs three memory reads to check whether the TPTL of the STT is empty, and jumps to the instructions implementing the normal virtual method dispatch if it is. If the TPTL is uninitialized, or non-empty, it checks the per-method cache of the running method. It jumps to the out of line slow path if the cache is uninitialized, the cache line for this shadow is uninitialized, or the cache line for this shadow is not empty. Otherwise it jumps to the instructions implementing the normal virtual method dispatch.

Instead of generating extra instructions at bytecodes corresponding to handler DJP shadows, we modified the VM's runtime exception handling routine to deal with handler DJPs, to more closely simulate a true interpreter.

As explained in Section 3.4 all applicable dispatchers are bundled up into a dispatcher caller which calls them around, before and after the DJP. We generate specialized dispatcher callers to efficiently handle the most common cases where only a single advice applies at a DJP and a generic dispatcher caller to handle the general case when more than one advice is applicable. All dispatcher callers which call any advice around or after the DJP execute the DJP from within their dynamic extent and skip execution of the original DJP when returning. Information needed to execute the DJP and skip over the instructions of the original shadow is passed to the out of line slow path when it is called. This includes an identifier indicating an `invokevirtual` bytecode as well as the size of the instructions of the normal virtual method call.

5. PERFORMANCE QUANTIFICATION AND ANALYSIS

We measure the performance of our implementation in two ways. The first exactly quantifies different micro operations in our implementation related to advice lookup and invocation. Specifically, we quantify the cost of executing advised and unadvised DJPs when following different paths through the interpreter. This allows us to compare the relative costs of the infrequently taken slower paths with the frequently taken faster paths. These benchmarks are discussed in Section 5.1 together with their results and analysis.

We also quantify the performance of our implementation with eight different AspectJ micro-benchmarks that allow us to compare the overall performance of our implementation to the ajc load time weaver. These benchmarks are discussed in Section 5.2 together with their results and analysis.

We use the statistically rigorous data collection methodology described by Georges et al. [10] and report the average running time of the various benchmarks over multiple invocations of the complete benchmark being measured. We compute 95% confidence intervals for each of the reported averages and we run each benchmark enough times to ensure that the 95% confidence intervals are no larger than 30ms. Although the length of the confidence intervals could be further reduced by repeating the benchmarks an increasing number of times, we found that a 30ms interval is sufficient to distinguish results from one another.

All of these micro-benchmarks are based on the same program setup and vary only in their use of pointcut and advice functionality. Each benchmark consists of a `SuperClass` which contains a method with 20 identical DJP shadows in a loop with a configurable number of iterations. Each of these shadows is a call to an alternate method which increments and tests a counter.[5]

Further, the `SuperClass` is extended by a `SubClass`, which is instantiated during the execution of the benchmark. Having both `SubClass` and `SuperClass` allows us to test both static and dynamic pointcut matching.

Each of the advice bodies also increments and tests a counter. The exact counter being incremented varies depending on the benchmark (it changes between incrementing a counter in the `SuperClass`, or the aspect itself). The specific differences between each benchmark and the general framework are discussed below.

Note that the 20 call DJP shadows give rise to a number of DJPs including 20 call and 20 execution DJPs among others. In fact, because the test method and advice body also contain field references for increments and tests, each iteration of the benchmark gives rise to 100 DJPs if no advice executes and 160 if there is one executed advice body. We have not included advice execution DJPs in these counts because our implementation currently does not support those.

Finally, all of the benchmarks are run on a 3Ghz dual core Intel Pentium 4 machine with 1GB of memory running Ubuntu Linux 8.10 with kernel 2.6.27. Since our implementation is based upon Jikes RVM version 2.9.2, we compare against an unmodified version of the same and ajc load time weaver version 1.5.2[6] in our benchmarks. Both VMs use a default

[4]Note that bytecode dispatch is a primary source of overhead in interpreters and compiling it out greatly improves execution performance.

[5]These micro-benchmarks are part of a suite we have also used for testing JIT optimized code [11]. The counter increment and test are designed to prevent the optimizing compiler from optimizing away the method bodies, calls, and possibly advice entirely.

[6]We had to modify the ajc load time weaver slightly to stop

heap size of 50MB.

Our implementation supports only the interpreter and not the JIT compiler in Jikes RVM, and so we disable recompilation using the JIT compiler for our data collection. Typical macro-benchmark will have code that gets recompiled with the optimizing JIT compiler, and disabling the JIT compiler means that measurements produced from the macro-benchmark are not accurate measurements of real-world performance. Instead, they measure a case that would never happen in a hybrid VM.

Given this drawback, we analyze our implementation with a suite of micro-benchmarks designed to quantify the micro performance of operations our implementation performs, and to micro measure startup performance.

5.1 Internal Micro-Benchmarks

The first set of micro-benchmarks we report were designed to measure the cost of different paths through the interpreter. The main goal of these benchmarks is to compare the cost of these paths to each other and to a baseline nearly empty method call.

All of these benchmarks were run with our RVM built in production mode, but configured to disable recompilation by the optimizing JIT compiler. We disable the JIT compiler for these benchmarks because we want to measure the relative costs of different paths through the advice weaver; to measure this we must execute many DJP shadows in a loop and this would normally trigger recompilation by the optimizer.

Each benchmark was run with a loop iteration count of 2^{15} except unadvised method (JikesRVM) and fastest fail (null TPTL) which use a 2^{23} iteration count. The iteration counts were selected to ensure the benchmark took at least three seconds to run. (Remember that each iteration results in at least 100 DJPs and up to 280 depending on the specific case, what advice runs etc.)

Figure 2 shows the results of running these seven micro-benchmarks on our implementation. This graph demonstrates the relative performance of the different paths through the interpreter. All the results are normalized to the baseline unadvised method benchmark.

Baseline method call (JikesRVM) measures the baseline cost of an unadvised call to a nearly empty method (the method just includes the counter increment and test). This benchmark was run on an identically configured plain RVM without our extensions.

Fastest fail (null TPTL) measures the fastest path to failure through the interpreter. This occurs when there is an empty TPTL for a given STT.

Fast fail (cache empty) measures the path through the interpreter where the TPTL is non-null, but the cache entry is empty (as opposed to cache not yet been constructed).

it from weaving its own classes and going into an infinite loop when run under Jikes RVM version 2.9.2

Cache hit simple measures the cost of executing a method call with a single matching `before` advice that has already been saved in the cache.

Cache hit complex I measures the cost of executing a method call with three applicable advice: two `before` advice with one of these having both a dynamic test on argument type and dynamic target type, and an `after` advice which also extracts the argument value from the call.

Cache hit complex II measures the cost of executing a method call with two applicable advice: a simple `before` advice and an `around` advice that has a dynamic match on a String argument and dynamic target type.

Cache miss fail measures the case in which there has not yet been a cache constructed and there is no matching advice. This and the next benchmark both require us to disable the cache filling behaviour of our implementation.

Cache miss match simple measures the case in which there has not yet been a cache constructed and a simple `before` advice matches, so this case includes the cost of building the dispatcher caller, dispatcher and invoking them to run the advice body.

Taken together, the two fast fail benchmarks measure an unadvised call DJP execution where the fact that no advice applies has already been confirmed and stored either in an empty TPTL or an empty cache entry. In the first and fastest case, the bar in Figure 2 is barely visible, but the annotation shows that the fast advice applicability check adds 28% overhead to the calling of a nearly empty method. In the second case it adds 134% overhead. These costs are accounted for by the additional loads and comparisons required to check the shadow cache. A comparative version of these cases appears in Section 5.2 and we compare this result to previously reported related work in Section 6.

The three cache hit benchmarks show the cost of going out of line, rechecking the cache and then running a dispatcher caller, one or more dispatchers and the applicable advice. The cache hit simple benchmark shows that a call DJP with one matching before advice executes roughly 10 times slower than the fastest fail case. Part of this overhead (roughly the cost of a baseline call or $1\times$) is accounted for by the execution of the advice itself. The remainder is due to the overhead of the dispatcher caller and dispatcher. (A small amount is due to construction of the TPTL and cache the first time through the benchmark loop, but that is amortized across the complete benchmark run and is an insignificant portion of the total time.) The cache hit complex I and II benchmarks are roughly 60 and 80 times slower than fastest fail. DJPs with more than one applicable advice execute much more slowly compared to simple cases because of the more complicated dispatching logic in the dispatcher. Further, complex II executes more slowly than complex I because of its use of `around` advice.

The cache miss fail benchmark shows the basic cost of going out of line, rechecking the TPTL and cache, together with the cost of building an empty cache and scanning the TPTL to find no matching advice. (Recording that there

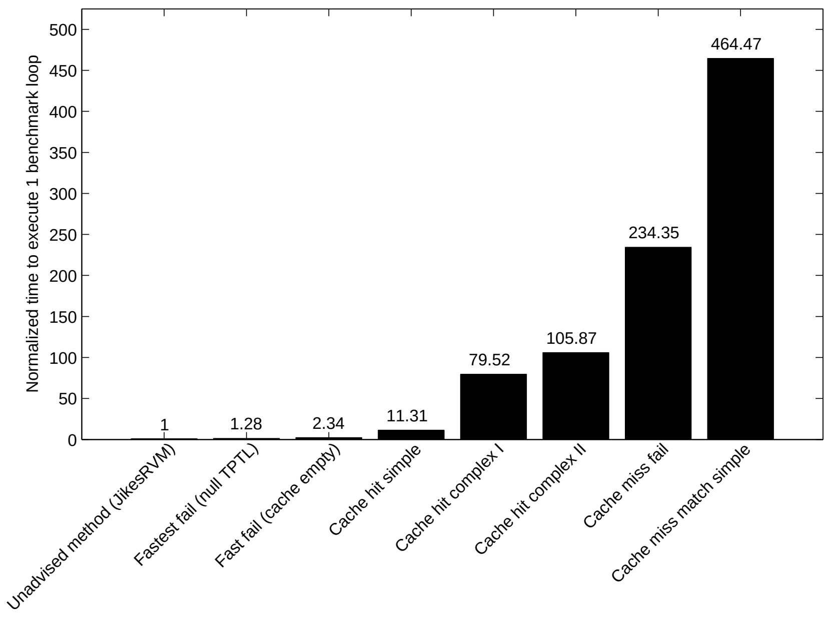


Figure 2: Quantitative benchmark results normalized to "Unadvised Method (JikesRVM)." Bars are annotated with their value.

is no matching advice in the cache is disabled to run the test.) Cache miss match simple is the same except it finds a simple matching advice and must build a dispatcher and dispatcher caller. The miss case is roughly 200 times slower than the fastest fail case and the match case is roughly 350 times slower. Note that in a normal execution (not running this micro-benchmark) these cases happens at most once per shadow because the presence or absence of matching advice is cached. Comparing the cache hit and miss cases shows the benefit of the per-method shadow cache.

5.2 Comparative Micro-Benchmarks

The second set of micro-benchmarks we report were designed to compare the performance of our interpreter based advice weaving implementation with the standard ajc load time weaver. The complete source code of these benchmarks can be downloaded from our website.[7]

Unadvised DJP Execution defines one advice which does not apply to any of the shadows in the benchmark loop. The advice body in this benchmark never runs.

Before Call DJP Execution defines a single `before` advice with a call pointcut which statically matches the call DJPs of all 20 shadows in the benchmark loop. The static match is definite – no dynamic residual testing is required.

Dynamic State Capture defines a `before` advice with a `call && args` pointcut in which the call statically matches all 20 shadows, and the args dynamically matches all 20 shadows (a residual test in the dispatcher). The advice body increments a counter in the object captured by the `args` pointcut.

[7]`http://www.cs.ubc.ca/~inaseer/oopsla/interp-bms.tar.gz`

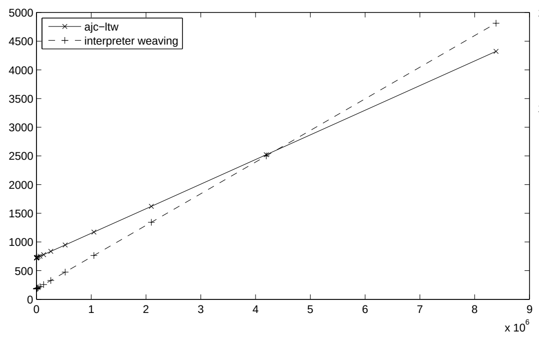
Around Advice defines a single `around` advice with a `call && args` pointcut. The entire pointcut matches all 20 shadows statically, and it extracts an `int` parameter from the DJP which is used in a `proceed` call in the `around` advice body.

CFlow w/o State Capture and *CFlow w/ State Capture* both define before advice with `call`, `execution` and `cflow` pointcuts. The first benchmark extracts no state for the advice, whereas the second uses a `this` pointcut to extract state from the context.

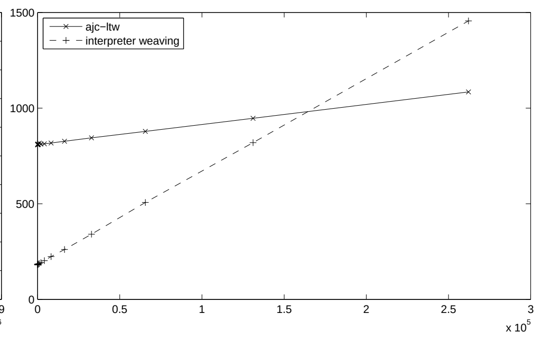
Multiple Advice involves multiple advice on the same DJP. It is a combination of three of the previous benchmarks. It incorporates the advice from Dynamic State Capture and Around Advice into one aspect, and has a second aspect which is the same aspect used in the Before Call DJP Execution benchmark. So in total, this benchmark defines three advice applicable at the 20 call DJP shadows.

Lots of Classes is the same as *Before Call DJP Execution* except that before the test loop is executed, the benchmark instantiates 100 different classes, each of around 1000 lines in length, in the constructor. None of the methods in these extra classes are executed.

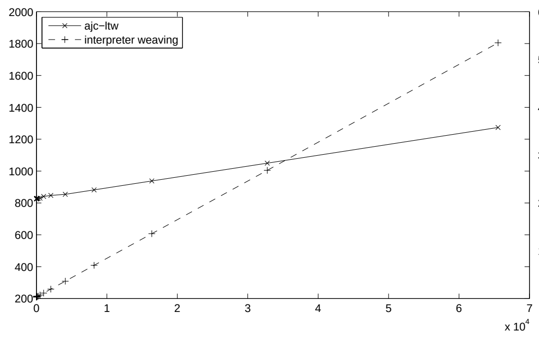
We run each of these benchmarks using a progressively increasing number of loop iterations $(2^0, 2^1, \cdots)$. For each number of loop iterations, the benchmark is invoked an appropriate number of times to get a 95% confidence interval which is no greater than 30ms, as discussed above. We gather this data for both our implementation and for ajc load time weaving. Both VM implementations are built in production mode and run with the optimizing compiler disabled.



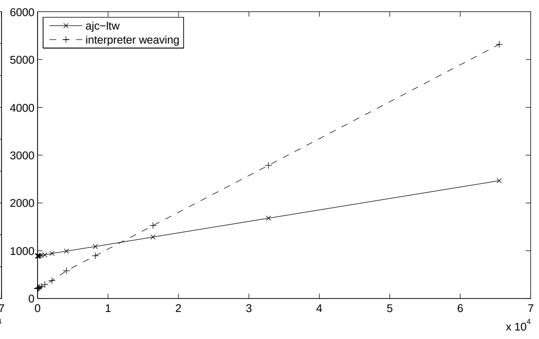
(a) Unadvised DJP Execution



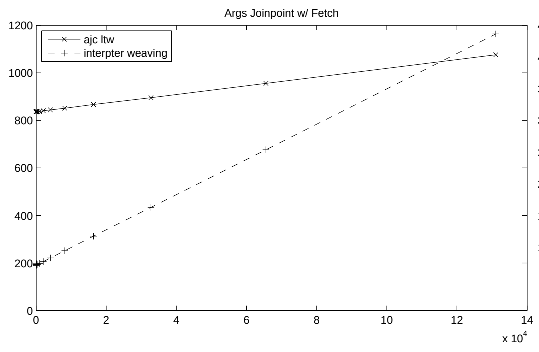
(b) Before Call DJP Execution



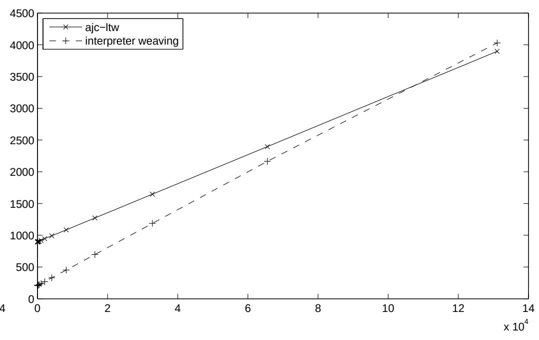
(c) CFlow w/o State Capture



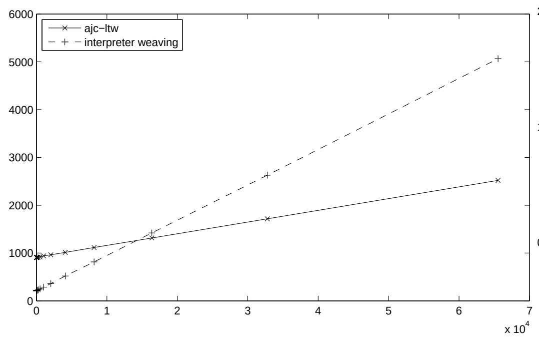
(d) CFlow w/ State Capture



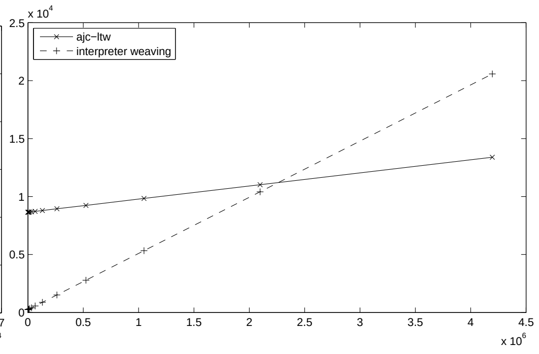
(e) Dynamic State Capture



(f) Around Advice



(g) Multiple Advice on DJP



(h) Lots of Classes

Figure 3: Comparative benchmark results. Number of iterations vs. execution time in milliseconds (scale varies).

Benchmark	to intersection		Slope interpreter/ajc
	Iterations	Time(ms)	
Unadvised DJP Execution	4,367,900	2600	1.2841
Before Call DJP Execution	164,520	980	4.6271
Dynamic State Capture	115,450	1050	4.0488
Around Advice	107,370	3350	1.2725
CFlow w/o State Capture	35,324	1067	3.5615
CFlow w/ State Capture	11,846	1177	3.2404
Multiple Advice	14,240	1262	3.0035
Lots of Classes	2,265,700	11208	4.2804

Table 2: Summary of comparison benchmarks. For each benchmark between each pair of performance curves we show the number of iterations of the benchmark required before the pair intersect, the elapsed time in milliseconds that it takes to get to the intersection point, and the slope of the interpreter weaving curve divided by the slope of the ajc curve.

Since the benchmarks are run with the JIT compiler disabled, the performance curves for both implementations are not an accurate account of the real world performance that these implementations would exhibit, because after some number of iterations of each benchmark, the JIT compiler would be invoked to optimize the executing method. If the JIT compiler were enabled, we would expect the performance curves reported by each benchmark to jump up at JIT time to account for the one iteration in which the JIT compiler was invoked, followed by another performance curve with a much shallower slope to reflect the improved performance after JIT optimization. (In a progressive optimization JIT, such as the adaptive optimization system in the Jikes RVM there could be several such jumps as the code is optimized further and further over time.)

The results of our performance measurements are shown in Figure 3. We graph the results with running time in milliseconds along the y-axis, and number of loop iterations along the x-axis. Please note scale differences between graphs. Only iterations of powers of 2 were actually computed, values between these results were linearly interpolated. We used linear interpolation here because we expected the running time to increase linearly in the number DJP executions we perform. By visual inspection we can see that this is indeed the case: the piecewise linear interpolation of the data yields nearly straight lines.

We report up to 2^{17} iterations for each of the benchmarks except for the Unadvised, Before Call, and Lots of Classes, for which we report 2^{23}, 2^{18}, and 2^{22} iterations respectively because the intersection points are further along.

Our RVM is graphed using a dashed line with `+` indicating measured values, and the ajc load time weaving implementation is graphed using a solid line and `x` marking the measured values.

All of our results of these benchmarks are also summarized in Table 2. In this table we report the number of iterations of the benchmark loop required before the curves cross over, the elapsed time in milliseconds that it takes to get to the cross over point, as well as the slope of our implementation's curve divided by the slope of ajc's curve. This last measurement shows, roughly, how many times faster a DJP execution is for ajc on the given benchmark.

We can make several observations from these eight benchmarks. First, as noted above, both implementations follow a linear curve proportional to the number of iterations (and hence the number of DJP executions). Additionally, we can see that, as we would expect, the slope of these curves is always greater for our implementation and shallower for the load time weaving based implementation. However, we can see from Table 2 that the slope of the interpreter weaving implementation performance curves change depending on which pointcut and advice may be applying at the given DJP.

Second, we can observe from the graphs that the interpreter weaving implementation always exhibits better first run (1 iteration) performance, and significantly better first run performance on the *Lots of Classes* benchmark. To understand this performance gain, consider the activities of the JVM on startup: in both cases there is the startup cost associated with the VM itself, and the cost of loading and executing the program. However, although not exactly quantified, in ajc load time weaving there is the additional cost of loading the weaver, and the cost of weaving all loaded application classes. In some small benchmarks, the cost of weaving application classes can be dominated by the cost of loading the weaver. However, the *Lots of Classes* benchmark shows that ajc's load time weaving has significant, inherent performance overhead loading application code if that code is not executed.

We instrumented IBM's Java VM using JVMTI to deterministically count the number of methods loaded and the number of methods executed during the startup of both Eclipse v3.5 and Tomcat v6.0.20[8]. Only around 43% and 46% of loaded methods were executed in each application respectively. These numbers suggest that the performance benefit gained by our implementation by avoiding the weaving of unexecuted code is significant.

5.3 Implications for Hybrid Implementations

Hybrid VMs will not improve the performance of all applications all of the time. Rather, the purpose of including an interpreter in these VMs is to balance the trade-off between startup performance and long-running, steady state

[8]Eclipse was run until the workbench appeared on the screen, and Tomcat was run until the output logs show "INFO: Server startup in xxxxx ms".

performance. The interpreter is intended to get the program executing quickly. The optimizing compiler is called to optimize frequently executed code. Furthermore, the interpreter should support the JIT compiler by providing it as much information as possible to reduce compile time, as well as to aid in improving the generated code quality.

Our implementation and benchmarks suggest that a hybrid implementation of AspectJ along these lines could meets these requirements in hybrid VMs. Our weaver exhibits good startup performance compared to ajc load time weaving. Further, for any method that is executed, interpreter weaving produces several data structures that could be used to improve the efficiency of per-method rewrite based weaving that would occur at JIT time. For any shadow that has executed, the per-method cache contains a dispatcher caller with all applicable advice; thus the advice matching work for such shadows is significantly reduced. A low level of JIT optimization could perhaps call the dispatcher caller directly and defer unfolding of the advice calls inline for a later optimization pass. For shadows that have not been executed, the TPTLs, TPTs and *.tnp list could be used to improve the performance of advice applicability matching.

Once a method has been JIT compiled, the per-shadow method cache can be garbage collected because it is no longer needed by the advice weaving interpreter. This releases the single most expensive data structure needed for interpreter based weaving.

Without a complete hybrid implementation, it is not possible to draw definitive conclusions about the potential benefits of a hybrid approach. But the relatively low interpreter overheads, the residual support for fast per-method weaving, and the release of the cache to be garbage-collected after rewriting based weaving leads us to conclude that building a complete hybrid implementation is warranted to support the full experiment.

6. RELATED WORK

The most commonly used AspectJ implementation is ajc [18]. As mentioned above, it is a bytecode rewriting based implementation. It has modes which allow it to perform rewriting at any stage bytecode is available outside the VM including compile time, post compile time, or load time.

The load time option (ltw) works by intercepting classes as they are being loaded by the VM and performing the rewriting at the interception point. This mode of operation is the one that most closely resembles our approach because only classes that are loaded are scanned for applicable DJP shadows. However, our approach goes further in that only DJP shadows that are actually executed are matched against, whereas ajc load time weaving will run the matcher against all shadows that are loaded. This allows our approach to have better application startup performance.

AspectWerkz[9] [6] implements an AspectJ-like language on top of traditional Java using a separate XML file for embedded annotations to specify advice and pointcuts in place of new language syntax. In other ways, Aspectwerkz functions much the same as ajc, although it supports dynamic weaving by hooking into VMs that support the Java Hotswap architecture.

[9]Note that circa January 2005, the ajc and AspectWerkz projects merged. Here, we refer to the most recent version of AspectWerkz before the merge.

The last related work that operates primarily on byte code is abc [1, 2, 3] which is an alternate compiler to ajc which focuses on optimizing generated code, and is built using an open, extensible compiler framework which allows it to be easily extended for research purposes. However, because it is an offline compiler, it also requires the scanning of all code for DJP shadows so that proper code rewriting can take place. Note, also, that abc does not support load time weaving, so the price of this scanning must be paid at compile time.

JAsCo [26, 27] is an implementation that is also based on bytecode rewriting, but it primarily uses a hook based approach in which generic hooks are inserted into the bytecode which call into JAsCo's advice dispatcher to handle the advice weaving. For optimization purposes, JAsCo also supports the ability to inline the advice dispatch code, as ajc and abc would, using Java VM compliant interfaces. This dichotomy is similar to the interpreter/JIT compiler dichotomy inside the VM, and so is similar to our approach in the abstract. However, our approach is integrated with the VM's interpreter rather than using bytecode rewriting to simulate interpreter-like lookup and invocation. Furthermore, like ajc, JAsCo requires all code to be rewritten with hooks to handle advice weaving. Our approach requires no additional scanning for advice weaving.

PROSE [24] was one of the first implementations which integrated AOP functionality into the VM; its implementation is based on the Jikes RVM. It implements an AspectJ-like language which supports much of the functionality of AspectJ but which also includes the ability to dynamically deploy aspects. Early versions of PROSE used the Java VM debugging interface to add hooks into the running program to effect advice dispatch, whereas more recent versions of PROSE integrated directly into the VM execution layers with two types of weaving: hook based weaving and runtime JBC weaving. Hook based weaving works on similar principles discussed above in that the code is rewritten with hooks at runtime to call into PROSE's runtime architecture to dispatch advice, and runtime JBC weaving works by weaving the VM's internal JBC representation directly at runtime. Although our approach and PROSE have similar conceptual implementations in the Jikes RVM, we differ from PROSE in that we do not rewrite JBC at all. We further optimize the common case by providing a number of differing weaving routines to support a fail fast strategy. Our table structures further facilitate efficient lookup and possible invocation of advice.

Steamloom [4, 14], like PROSE, integrates advice weaving into the VM, and is based on the Jikes RVM. Steamloom implements a fully dynamic AspectJ-like AOP language by integrating a runtime rewriting based weaver which rewrites the VM's internal representation of the JBC. This weaver can weave and unweave advice execution logic to deploy and undeploy aspects at runtime. Steamloom's load time and weaving architecture has not been optimized for efficient load time or first run performance; it currently uses a memory intensive representation for access to the internal representation of the bytecode for easy manipulation and rewriting. It has been optimized for steady state performance, whereas our implementation has been optimized for startup performance.

The Nu [9, 8] VM resembles our approach in that it augments the interpreter to effect advice weaving. Nu extends the Java VM with two additional bytecodes which control a VM interval weaver. The implementation is based on the Sun Hotspot VM and supports only the interpreter. It uses the *point-in-time* join point model [21] with a dynamic language semantics and method execution and return dynamic join points. The machine code for our fastest fail path is nearly identical to the corresponding machine code in Nu, but the table and cache structures following those paths are significantly different.

ALIA [5] is an interface designed to more completely separate frontend compilers and VM internal weaving mechanisms to support runtime weaving and dynamic deployment of aspects. It makes AOP language constructs into first class entities inside the VM to allow user programs to have a consistent interface to the VM for controlling the runtime weaver. Our work and ALIA are largely orthogonal: although we only support ajc's load time weaving interface for advice weaving inside the VM, our architecture should be largely applicable to alternate VM interfaces including ALIA. One main difference that would make our work harder to integrate is that our design is not intended to be used for dynamic aspect deployment.

7. FUTURE WORK

For simplicity we key advice lookup on a single DJP shadow property – the static type of the target. One area of future work would be to explore the possible benefits of having different kinds of DJPs key on different properties. A related issue is to explore the trade-offs associated with using a more compact per-method shadow cache.

Another avenue of future work will be to support the complete AspectJ language including support for all kinds of DJPs, `thisJoinPoint` and all the aspect instantiation models. The architecture presented in this paper requires each DJP shadow to have a STT from which it can fetch the TPTL. Advice execution DJPs do not have a STT and will therefore require a small addition to the architecture. We plan to maintain a separate list of advice that can match at advice execution DJPs. The advice in this list will have to be checked and potentially invoked before any dispatcher invokes an advice.

Furthermore, we made our comparisons against an external load time weaving implementation: ajc load time weaving. A more appropriate comparison could be made against a load time weaving based approach that is fully integrated into the VM and optimized. Unfortunately such an implementation does not exist for comparison (several implementations of load time weaving like approaches do exist, or could be simulated using existing approaches [11, 4], but none of these implementations have been optimized for startup efficiency). One avenue of future work we intend to pursue is to produce an optimized load time rewriting based weaver.

Finally, without a complete hybrid implementation supporting mixed mode execution, we can only speculate on the overall macro and micro performance of the implementation. So, another avenue of future work is to explore extending our presented architecture and implementation to include the optimizing JIT compiler of the Java VM. It is only in the context of a complete hybrid implementation can we make definitive conclusions about the utility of interpreter weaving and load time weaving.

8. CONCLUSIONS

We have proposed an architecture for Java VM interpreters to support efficient advice weaving. The architecture has been designed to support a fast fail strategy by recording that no advice can apply at a shadow through an empty TPTL or an empty cache entry.

We presented an implementation of this architecture based on the Jikes RVM. Although the Jikes RVM does not contain an actual interpreter, we argue that the use of the Jikes RVM baseline compiler as a substitute for an actual interpreter is sufficient for the experiments we conducted, and so our results should also apply to pure interpreters.

We analyzed our implementation through two sets of benchmarks: the first quantifies the relative performance of different paths through the interpreter and the second compares our implementation against the ajc load time weaver using the Jikes RVM. The results of these benchmarks show that there is reason to believe that a complete hybrid implementation using our interpreter architecture could provide good overall performance. In particular, the startup performance is better than an external load time rewriting approach, and the execution efficiency is still reasonable.

9. ACKNOWLEDGEMENTS

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

10. REFERENCES

[1] ABC Group. abc (AspectBench Compiler). `http://aspectbench.org`.

[2] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhohák, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, 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.

[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 and Mira Mezini. A Flexible Architecture for pointcut-advice Language Implementations. In *VMIL '07: Proceedings of the 1st workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms*, page 1, New York, NY, USA, 2007. ACM.

[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 jalapeÃśo 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] Robert Dyer and Hridesh Rajan. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 191–202, New York, NY, USA, 2008. ACM.

[9] Robert Dyer, Rakesh B. Setty, and Hridesh Rajan. Nu: Toward a flexible and dynamic aspect-oriented intermediate language model. Technical report, Iowa State University, June 2007.

[10] Andy Georges, Dries Buytaert, and Lievan Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, 2007.

[11] Ryan M. Golbeck, Samuel Davis, Immad Naseer, Igor Ostrovsky, and Gregor Kiczales. Lightweight Virtual Machine Support for AspectJ. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 180–190, New York, NY, USA, 2008. ACM.

[12] Ryan M. Golbeck and Gregor Kiczales. A Machine Code Model for Efficient Advice Dispatch. In *VMIL '07: Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 2, New York, NY, USA, 2007. ACM Press.

[13] ABC Group. Aspectj benchmarks. `http://abc.comlab.ox.ac.uk/benchmarks`.

[14] 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.

[15] 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.

[16] J. Collins-Unruh and G. Murphy. Aspect-oriented jEdit. Unpublished.

[17] JBoss AOP Team. Framework for Organizing Cross Cutting Concerns. `http://www.jboss.org/jbossaop/`.

[18] 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.

[19] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[20] Marius Marin, Leon Moonen, and Arie van Deursen. An Integrated Crosscutting Concern Migration Strategy and its Application to JHoTDraw. Technical report, Delft University of Technology, Mekelweg, Delft, The Netherlands, 2007.

[21] Hidehiko Masuhara, Yusuke Endoh, and Akinori Yonezawa. A Fine-Grained Join Point Model for More Reusable Aspects. In *APLAS 2006: Proceedings of the Fourth ASIAN Symposium on Programming Languages and Systems*, pages 131–147, November 2006.

[22] IBM J9 Team Members. personal communication, 2009.

[23] 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.

[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.

[25] SpringSource.org. `http://www.springframework.org/`.

[26] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[27] Wim Vanderperren, Davy Suvée, Bart Verheecke, María Agustina Cibrán, and Viviane Jonckers. Adaptive programming in jasco. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM.

[28] Alexandre Vasseur, Jonas Bonér, and Joakim Dahlstedt. Jrockit jvm support for aop, part 2. `http://www.oracle.com/technology/pub/articles/dev2arch/2005/08/jvm_aop_2.html`.