Interpreter Implementation of Advice Weaving

by

Muhammad Immad Naseer

B.S., National University of Computer and Emerging Sciences, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2009

Abstract

Implementing advice weaving using code rewriting can cause performance problems during application startup and incremental development.

We present an interpreter based (non-rewriting) approach using 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 in the JikesRVM, an open source research virtual machine, shows the approach is straightforward to implement. Internal micro-benchmarks show dynamic join point execution overhead of approximately 28% in the common case where no advice is applicable. Comparative micro-benchmarks with ajc load time weaving show startup time reductions equivalent to the interpreted execution of 100-117 million unadvised dynamic join points. 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.

Table of contents

List of tables

List of figures

Acknowledgements

First and foremost, I would like to thank my supervisor, Gregor Kiczales, for his guidance, patience and support during the course of this work. I am highly indebted to him for always pushing me to do better and for his useful and instructive criticism — it made me learn and improve a lot more than I would have otherwise.

I would like to thank Ryan Golbeck for his helpful advice and support throughout this project. I would also like to thank Sam Davis for his contributions to the project. This thesis is based on a paper I co-authored with Ryan, Gregor and Sam and owes a lot to their valuable contributions.

I would also like to thank Gail Murphy for being my second reader and providing helpful comments on this work.

Finally, I thank my family, especially my parents, for their encouragement and understanding throughout my studies at UBC. This would not have been possible without their loving support.

Chapter 1

Introduction

Pointcut and advice functionality is an important part of several aspect-oriented programming (AOP) [22] languages and toolkits, including AspectJ [21], CaesarJ [26], JBoss AOP [20], and Spring [30]. 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 [18, 2]; at load time using bytecode rewriting [21, 14]; during runtime using bytecode rewriting, including the ability to do dynamic deployment of aspects with advice [5]; and during just-in-time (JIT) compilation by augmenting the machine code generation process [15]. Work on Spring, JBoss, JAsCo, JRockit, PROSE, and Nu [30, 20, 31, 34, 28, 10] has implemented advice weaving using runtime lookup and invocation.[1]

Each DJP in a program's execution has a corresponding *DJP shadow* in the program text [18]. In rewriting approaches, DJP shadows are augmented with additional code to execute advice bodies appropriately. Approaches to weaving that involve code rewriting prior to execution effectively compile out the static overhead of advice lookup. This offers the advantage of better runtime performance, but it has the corresponding disadvantage of requiring a global scan of the code in order to find potentially advised shadows and rewriting code at all 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.

While ajc has an incremental reweaving mode intended to better support interactive development, it does not solve a related problem that arises using 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

[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.

the VM, whether it is executed or not. This can significantly impair startup performance.

The work presented here is part of a project exploring whether a *hybrid implementation strategy* for advice weaving can provide a good balance between quick start up and optimal steady state execution efficiency. By hybrid we mean that just as Java implementations combine an interpreter and JIT compiler, 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 thesis addresses one key part of the 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 interpreter implementations can be straightforwardly extended to work this way; (iv) none of the static compiler, loader or interpreter must perform a global scan typical of rewriting based implementations; and (v) the interpreter leaves behind information which can be used to facilitate rewriting based weaving performed when and if code is later JIT compiled.

The contributions of this thesis are: a simple analysis of advice frequency in AspectJ programs; a design for interpreter advice weaving; an implementation in the Jikes RVM; and a performance analysis that suggests the performance is good enough to warrant development of a complete hybrid implementation. In particular, for a set of appropriate micro-benchmarks, the startup time of our implementation is 600-700ms faster than for ajc's load time weaving (on a 3Ghz dual core Intel Pentium 4), and it leaves behind information that could be used by later (JIT based) per-method rewriting.

Chapter 2

Advice applicability in AspectJ programs

The relative frequency of advice being applicable at DJPs is an important factor in our design. Table 2.1 shows advice applicability information gathered from two applications (AJHotDraw [24] and aspectJEdit, a version of jEdit refactored using AspectJ [19]); three of the AspectJ example programs (telecom, observer and tracing); and 14 of the abc benchmarks [16].[2] As the 14 benchmarks are not actual applications, they reflect idiosyncratic behavior typical of benchmarks. 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 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 in table 2.1 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 data in table 2.2 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]

[2] 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.

	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
ants	21	14	8	–	–	–	–	–	11	5
cona-sim	4	< 1	5	< 1	3	< 1	–	–	4	< 1
cona-stack	11	48	10	3	6	18	–	–	9	25
dcm-sim	–	–	88	50	–	–	–	–	11	< 1
nullcheck-sim	30	28	–	–	–	–	–	–	10	4
nullcheck-sim_after	–	–	19	28	–	–	–	–	3	4
nullcheck-sim_notwithin	26	28	–	–	–	–	–	–	9	4
nullcheck-sim_orig	69	99	–	–	–	–	–	–	23	12
quicksort_gregor	22	39	–	–	–	–	–	–	9	15
quicksort_oege	22	39	–	–	–	–	–	–	9	15
bean	6	7	–	–	–	–	–	–	3	3
bean_gregor	6	12	–	–	–	–	–	–	3	4
figure	22	47	–	–	–	–	–	–	7	15
hello	30	28	–	–	–	–	–	–	10	10
Overall	4	13	5	< 1	< 1	< 1	< 1	< 1	3	4

Table 2.1: 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 remainder are abc benchmarks. 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.

	call		execution		get, set		other	
AJHotDraw	54	44	16	25	22	24	8	7
aspectJEdit	51	21	9	13	35	65	5	1
observer-project	33	40	16	19	27	33	24	8
telecom-project	43	40	18	20	23	28	15	12
tracing-project	48	56	18	23	15	18	19	3
ants	46	33	15	26	28	37	11	3
cona-sim	49	14	11	13	30	71	9	< 1
cona-stack	65	40	8	27	19	27	8	6
dcm-sim	34	32	13	2	41	65	12	2
nullcheck-sim	33	13	15	13	42	69	10	4
nullcheck-sim_after	33	13	15	13	42	69	10	4
nullcheck-sim_notwithin	33	13	15	13	42	69	10	4
nullcheck-sim_orig	33	12	15	12	42	63	10	12
quicksort_gregor	40	39	12	16	24	30	24	15
quicksort_oege	40	39	12	16	24	30	24	15
bean	54	47	16	22	19	26	10	5
bean_gregor	52	36	17	26	20	31	12	7
figure	29	32	23	32	24	32	24	4
hello	32	38	19	13	19	27	29	22
Overall	50	32	12	22	31	44	7	3

Table 2.2: The ratio of different kinds of shadows and DJPs to the total number of shadows and DJPs. For each kind of DJP, the left sub-column is the percentage of shadows of that kind and the right sub-column is the percentage of DJPs of that kind. The *other* column includes initializer, static initializer, preinitializer, handler and advice execution DJPs. The *Overall* row shows the percentages of shadows and DJPs of the corresponding kinds taken over all input programs.

Chapter 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, including modifiers, the static type of the target (STT), the target name (TN) and others. All kinds of DJPs except advice execution have an 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

AspectJ does not define a dynamic deployment semantics for advice. While there have been proposals for this [26, 20], 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 3.1, when advice declarations are loaded the pointcuts are processed 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 the conservative bound over each value in the triples. As part of the abstract evaluation user defined named pointcuts are expanded inline. The abstract evaluation rules are shown in Figure 3.2. Note that we implement the intersection of type and name pat-

```
class C {
  void m() {..}
}
```

```
class Sup {
  void m() {..}
  void mSup() { .. }
}

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

```
--- advice ---

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

```
--- triples ---

<{call} {C} {m}>
<{call} {Sup} {mSup}>

<{call} {Sup} {mSup}>
<{call} {Su*} {mSu*}>
<{call} {*} {*}>
<{exec} {*} {mSu*}>
```

```
-- STT Type Pattern Tables (TPTs) --

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

```
-- *.tnp List --

   (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 3.1: A simple example and the corresponding lookup table structures. 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.

terns as a union. A better strategy would be to merge any patterns that can be merged or return an empty set if some patterns cannot possibly occur together. We leave this optimization as future work.

Once the advice pointcut has been abstractly evaluated the advice declaration is added to a set of *STT type pattern tables* (TPTs)[4] 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 3.1 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 3.1 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

[4] We abbreviate STT type pattern table to TPT instead of STTTPT because the former is unique and TLAs are easier to remember.

$\mathbf{call}(\text{any } tp.np(\text{any})) \rightarrow < \{\mathbf{call}\}, \{tp\}^5, \{np\} >$
$\mathbf{exec}(\text{any } tp.np(\text{any})) \rightarrow < \{\mathbf{exec}\}, \{tp\}, \{np\} >$
$\mathbf{get}(\text{any } tp.np) \rightarrow < \{\mathbf{other}\}, \{tp\}, \{np\} >$
$\mathbf{set}(\text{any } tp.np) \rightarrow < \{\mathbf{other}\}, \{tp\}, \{np\} >$
$\mathbf{handler}(tp) \rightarrow < \{\mathbf{other}\}, \{tp\}, \{*\} >$

$\mathbf{!}(\text{A}) \rightarrow [\text{not } A]$
$\mathbf{!}\,[\text{not } A] \rightarrow A$

$\mathbf{within}(tp) \rightarrow < \{*\}, \{*\}, \{*\} >$
$\mathbf{withincode}(\text{any } tp.np(\text{any})) \rightarrow < \{*\}, \{*\}, \{*\} >$
$\mathbf{cflow}(pc) \rightarrow < \{*\}, \{*\}, \{*\} >$
$\mathbf{cflowbelow}(pc) \rightarrow < \{*\}, \{*\}, \{*\} >$
$\mathbf{target}(tp) \rightarrow < \{*\}, \{*\}, \{*\} >$
$\mathbf{this}(tp) \rightarrow < \{*\}, \{*\}, \{*\} >$
$\mathbf{args}(tp\ list) \rightarrow < \{*\}, \{*\}, \{*\} >$
$\mathbf{if}(boolean\ exp) \rightarrow < \{*\}, \{*\}, \{*\} >$

$\&\&([\text{not } A], [\text{not } B]) \rightarrow [\text{not or}(A, B)]$
$\&\&([\text{not } A], B) \rightarrow B$
$\&\&(A, [\text{not } B]) \rightarrow A$
$\&\&(A, B) \rightarrow \text{and}(A, B)$

$||([\text{not } A], [\text{not } B]) \rightarrow [\text{not and}(A, B)]$
$||([\text{not } A], B) \rightarrow < \{*\}, \{*\}, \{*\} >$
$||(A, [\text{not } B]) \rightarrow < \{*\}, \{*\}, \{*\} >$
$||(A, B) \rightarrow \text{or}(A, B)$

$and(< A_k, A_{tp}, A_{np} >, < B_k, B_{tp}, B_{np} >) \rightarrow < A_k \cap_k B_k, A_{tp} \cap_t B_{tp}, A_{np} \cap_n B_{np} >$

$or(< A_k, A_{tp}, A_{np} >, < B_k, B_{tp}, B_{np} >) \rightarrow < A_k \cup_k B_k, A_{tp} \cup_t B_{tp}, A_{np} \cup_n B_{np} >$

$\{*\} \cap_k B \rightarrow B$
$A \cap_k \{*\} \rightarrow A$
$A \cap_k B \rightarrow A \cap B$

$\{*\} \cap_t B \rightarrow B$
$A \cap_t \{*\} \rightarrow A$
$A \cap_t B \rightarrow A \cup B$

$\{*\} \cap_n B \rightarrow B$
$A \cap_n \{*\} \rightarrow A$
$A \cap_n B \rightarrow A \cup B$

$\{*\} \cup_k B \rightarrow \{*\}$
$A \cup_k \{*\} \rightarrow \{*\}$
$A \cup_k B \rightarrow A \cup B$

$\{*\} \cup_t B \rightarrow \{*\}$
$A \cup_t \{*\} \rightarrow \{*\}$
$A \cup_t B \rightarrow A \cup B$

$\{*\} \cup_n B \rightarrow \{*\}$
$A \cup_n \{*\} \rightarrow \{*\}$
$A \cup_n B \rightarrow A \cup B$

Figure 3.2: Abstract evaluation rules for pointcuts. Keywords in bold correspond to primitive AspectJ pointcuts. The upper section shows the evaluation rules for leaf pointcuts and **!**. All of these evaluation rules return either a triple containing a set of pointcut kinds, a set of STT type patterns, and a set of TN name patterns, or they return *not* of such a triple. The middle section shows the combination rules for **&&** and ||. The lower section shows the rules for combining each value in the triples using the **and** and **or** subroutines. Note that since the abstract evaluation may contain a residual *not*, the last step of the algorithm converts [not A] to $< \{*, *, *\} >$ if necessary.

spread of a pointcut such as in advice 6 in Figure 3.1 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 [18]. 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. The fast fail path code is responsible for trying to quickly prove that no advice applies at the shadow and if so, proceeding to execute the VM instructions corresponding to the shadow normally. When the absence of applicable advice cannot be proved quickly the fast fail path goes to an 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 (neither empty nor non-empty) 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.

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 efficiency, 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

[5] 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.

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 [18].

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.

Chapter 4

Implementation

To evaluate our interpreter weaving architecture we implemented it using version 2.9.2 of the Jikes Research Virtual Machine (RVM) [9]. Our implementation supports all five kinds of advice and all DJPs except initialization, pre-initialization, 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.

The RVM does not have a true interpreter – it uses a simple *baseline compiler* in the place of an interpreter. The baseline 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. In addition, some values which a true interpreter would fetch from the bytecode are copied into the generated machine code because the machine code does not have access to the bytecode when it runs.

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. 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. In working with the baseline compiler we can reasonably consider ourselves to be developing an interpreter as long as we follow the same coding discipline as in the baseline compiler.

4.1 Instructions executed inline at DJPs

We modified the baseline compiler to generate extra instructions for weaving advice at DJP shadows. These instructions are responsible for quickly proving that no advice applies at the DJP and continuing with the normal execution in that case. They call an out of line slow path if the applicability of advice cannot be ruled out.

Figure 4.1 shows the machine code generated for call DJP shadows at `invokevirtual` bytecodes. The 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.

In a pure interpreter, the code for dealing with method execution DJPs will have to be placed at the beginning of the interpreter dispatch loop. Since constructor and advice bodies are also translated into regular Java methods and would be run by the same dispatch loop, it would need to distinguish normal methods from advice and constructors so that it does not erroneously treat the start of their execution as method execution DJPs. While such a check is not required in the baseline compiler as it knows the method it is generating the machine code for, we still generated the machine code for the check in order to faithfully model the behavior of an interpreter. The check was placed after the TPTL and cache check so that it does not slow down the cases that follow the fast fail path.

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

4.2 Dispatchers

The dispatchers are responsible for invoking the advice, fetching and passing in any arguments expected by the advice and guarding its invocation with any residual dynamic checks.

We implemented a number of different dispatchers to deal with different combination of advice and pointcuts. As shown in table 4.1, the different dispatchers can be understood as a cross product of three different axes of variation among the cases:

- The locations where the DJP context needs to be fetched from.
- The registers into which some of the arguments have to be copied.
- Whether the invocation of the advice has to be guarded with residual dynamic checks.

```
 ; -- check whether TPTL is null --
 mov eax, jtoc[9776]        ; load VM_TypeReference.types array
 mov edx, eax[8204]         ; load VM_TypeReference for STT
 mov edx, edx[12]           ; load VM_TypeReference.TPTL field
 cmp edx, 0                 ; if TPTL is null
 je normal_call             ;   its a normal call (fastest fail)

 ; -- cache check --
 mov edx, jtoc[9524]        ; load VM_MemberReference.members array
 mov edx, edx[113520]       ; load VM_MemberReference of caller
 mov edx, edx[20]           ; load VM_MethodReference.resolvedMember
 mov edx, edx[18]           ; load actual cache vector
 cmp edx, 0                 ; is vector not allocated?
 jne alloctd                ; if vector is allocated go check it
 je go_out_of_line          ; otherwise go out of line

alloctd:
 mov edx, edx[52]           ; load cache entry for the bytecode index
 cmp edx, jtoc[7362]        ; if cache entry is empty
 je normal_call             ;   its a normal call (fast fail)


go_out_of_line:
 push esp                   ; stack pointer
 push esi[96]               ; frame pointer
 push 3                     ; indicates this is an invokevirtual
 push 28380                 ; caller id
 push 28379                 ; callee id
 push 12                    ; byte code index
 push 18                    ; #size of normal call instructions
 mov eax, esp[24]           ; load first parameter
 mov edx, esp[20]           ; load second parameter
 call jtoc[7972]            ; call out of line slow path

normal_call:
 mov edx, esp[8]            ; load object
 mov ecx, edx[-12]          ; load object's TIB
 mov eax, esp[8]            ; load first parameter
 mov edx, esp[4]            ; load second parameter
 call ecx[68]               ; call virtual method
 push eax                   ; push return value on stack
```

Figure 4.1: Machine code to interpret a call DJP shadow (invokevirtual instruction). Code with a vertical line in the margin is the fastest fail path. Values in italics are offsets and other values from the bytecode which the baseline compiler architecture copies into the machine code; in a true interpreter this machine code would instead fetch these values from the bytecode. In our actual implementation, we reduce code bloat by having the baseline compiler emit copies of only the check TPTL and normal call segments for each shadow; the remainder of this code is implemented as a machine code subroutine.

No context, filling EAX
No context, filling EAX, test residual pointcuts
No context, filling EAX and EDX
No context, filling EAX and EDX, test residual pointcuts
SP relative, filling EAX
SP relative, filling EAX, test residual pointcuts
SP relative, filling EAX and EDX
SP relative, filling EAX and EDX, test residual pointcuts
SP relative, fill all required registers
SP relative, fill all required registers, test residual pointcuts
FP relative, filling EAX
...
SP and FP relative, filling EAX
...
Variable SP relative, filling EAX
...
Variable SP and FP relative, filling EAX
...
Generic, filling EAX
...

Table 4.1: The different dispatchers generated for dealing with different combinations of advice and pointcuts. Note that the 'No context' family of dispatchers does not have a 'fill all required registers' case as they can only be passed a maximum of two parameters, the aspect instance and the around closure object. All the other family of dispatchers contain all the cases as shown for the 'SP relative' family of dispatchers.

4.2.1 Advice invocation

The ajc compiler translates advice bodies into normal Java methods taking the DJP context captured by pointcuts as method arguments. The dispatchers are responsible for fetching the DJP context captured by the pointcuts and calling the advice method following the JikesRVM's method calling conventions. The RVM's method calling conventions vary from architecture to architecture; we implemented our extensions to the RVM only for the IA32 architecture.

Fetching the DJP context

We implemented a number of specialized dispatchers to fetch the DJP context efficiently for the common simple cases and a number of generic dispatchers for the rest of the cases. The specialized dispatchers use an array of indices to fetch the arguments with the minimum of computation at runtime without any need to interpret the pointcut. The generic dispatchers have to interpret the pointcut in order to push the required values on the stack and thus execute slowly compared to the specialized versions.

We describe the different cases below along with example advice which leads to the construction of dispatchers implementing these cases.

No context

This is the simplest case where the advice does not capture any join point context. The dispatchers only have to push the advice instance and possibly an around closure object on the stack.

```
before(): call(* C1.m1(..)) { ... }
```

SP relative

This case handles advice whose captured DJP context can be fetched from constant indices relative to the stack pointer.

```
before(C1 c1, String s): set(* C1.f1)
                         && args(s)
                         && target(c1) { ... }
```

FP relative

This case handles advice whose captured DJP context can be fetched from constant indices relative to the frame pointer.

```
before(C1 c1, String s): execution(* C1.m1(..))
                         && args(s)
                         && this(c1) { ... }
```

SP and FP relative

This case handles advice whose captured DJP context can be fetched from constant indices relative to the stack and frame pointer. The indices relative to the FP are stored as negative integers.

```
before(C2 c2, String s): call(* C1.m1(..))
                         && args(s)
                         && this(c2) { ... }
```

Variable SP relative

This case handles advice whose captured DJP context can be fetched at constant indices relative to the stack pointer given the total words taken on the stack by the arguments of the DJP. Since the number and size of arguments varies from DJP to DJP, the exact indices the arguments are fetched from also varies from DJP to DJP.

```
before(String s): call(* C1.m1(..))
                  && args(s, ..) { ... }
```

Variable SP and FP relative

This case is similar to the above except that it except that some arguments are fetched at constant indices relative to the frame pointer. The indices relative to the frame pointer are stored as negative integers.

```
before(C2 c2, String s): call(* C1.m1(..))
                         && args(s, ..)
                         && this(c2) { ... }
```

Generic

This case handles the rest of the advice, including advice which captures context using the cflow pointcut.

```
before(C1 c1, String s): call(* C1.m1(..))
    && target(s)
    && cflow(call(* C2.m2(..)) && args(s)) { ... }
```

Finally note that we select dispatchers based solely on the advice without taking any information about the DJP shadow at which it applies into account. Taking information about the advised DJP into account can not only remove the need for the variable SP cases but also allow for more cases to be handled by the specialized dispatchers instead of being delegated to the generic dispatchers. We leave this optimization as future work.

Copying parameters into registers

The JikesRVM's method calling conventions for the IA32 architecture require the first two non-floating point parameters to be copied into `EAX` and `EDX` registers respectively, and the first several floating point parameters to be copied into the special purpose floating point registers.

We implemented three different kinds of dispatchers to handle the following three cases: the parameter just has to be copied into the `EAX` register; the parameters just have to be copied into the `EAX` and `EDX` registers; and all the other cases.

4.2.2 Dynamic checks

The dispatchers are responsible for guarding the invocation of advice with any residual dynamic checks. 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 dispatchers delegate the dynamic tests to be performed to a *dynamic check object*. As shown in table 4.2, we generate different dynamic check objects for fetching the values from different parts of the stack and testing them to see if they have the right type, similar to how we generate different dispatchers for different cases. We generate specialized dynamic check objects

SP relative
FP relative
SP and FP relative
Variable SP relative
Variable SP and FP relative
Generic

Table 4.2: The different dynamic check objects generated for different dynamic residual pointcuts. The specialized dynamic check objects are only generated for residual pointcuts strung together by && pointcuts; the rest of the cases are handled by the generic dynamic check object.

only for those dynamic residuals that contain pointcuts strung together by && pointcuts and fall back to the generic dynamic check object for the other cases.

4.3 Dispatcher callers

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 case 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. The generic dispatcher is also responsible for dealing with advice precendence and calling the dispatchers before, after and around the DJP in the right order.

4.4 Executing DJPs

All dispatcher callers which call the advice around or after the DJP execute the DJP from within their dynamic extent and skip over the original DJP shadow when returning from the slow path.

All the information needed to execute the DJP within the dynamic extent of the dispatcher caller and to skip over the original DJP shadow is passed to the out of line slow path when it is called. As figure 4.1 shows, we pass in the stack pointer, the frame pointer, the current bytecode, and the ids for the VM internal method object for the caller and callee to the out of line path. This information is sufficient for the dispatcher caller to reconstruct the stack frame by copying all the arguments from below the stack, copying

the relevant arguments into registers and performing the virtual method dispatch. We also pass in the size of the instructions of the normal virtual method call so that the dispatcher caller can add that to the return address when returning from the slow path thus skipping over the original shadow if required.

We implemented the execution of `get` and `set` DJPs in a manner very similar to that described for `call` DJPs above. Our implementation of the `execution` DJPs is different from the rest however and deserves more explanation. Execution DJPs call the out of line path before the execution of the bytecodes begins but after the method prologue responsible for setting up the stack frame has run. In order to execute the execution DJPs from within the dynamic extent of the dispatcher callers, we first copy the stack frame of the original method from below the stack. We then modify the return address on the copied stack frame so that the method returns to the dispatcher caller instead of to its original caller. We continue the execution of the method by jumping to the return address of the out of line path. In order to prevent execution of the method for a second time when returning from the out of line path, we return directly to the original caller of the method instead of returning to the method.

Note that we did not have to implement support for executing `handler` DJPs as only `before` advice is supported at those DJPs[18].

Though we wrote custom machine code to execute the DJPs instead of relying on the VM's reflection capabilities, the DJPs still run slower as compared to the original instructions because the dispatcher callers have to perform a number of checks to decide which machine code snippet to execute.

4.4.1 Proceed statement

The ajc compiler translates proceed statements into calls to a special synthetic method. We modified the baseline compiler to translate those calls to calls to a VM internal proceed method. A similar effect can be achieved in a pure interpreter by tagging the `invokestatic` instruction calling the synthetic proceed method so that it is handled by a different branch in the interpreter dispatch loop. This tagging will require the VM to scan the bytecodes of the around advice looking for `invokestatic` instructions calling the proceed method. Modern VMs already scan the bytecode prior to method execution – such as for bytecode verification[23] or for translating bytecodes into a form that can be used by a threaded interpreter[12] – and this search and tagging operation can be integrated with one of those scans.

Ajc generated around advice bodies take an *around closure object* which

the proceed method is supposed to use to invoke the underlying DJP. We use the around closure object to pass in information needed by the VM internal proceed method to invoke the underlying DJP. The proceed method uses the same code as that used by the dispatcher callers to execute the underlying DJP. In order to minimize the object creation overhead, the dispatcher callers use a pool of around closure objects instead of creating a new one when calling an around advice each time. Finally, since the proceed statement can be used to change the DJP context values captured by the around advice, the VM internal proceed method overwrites the DJP context values on the stack with those passed to the proceed method before invoking the next advice or underlying DJP so that they see the new values.

4.5 Interaction with the VM

Our implementation interacts with the VM at a small number of well defined places. These include parts of the interpreter that execute DJP shadows, the class loader to process advice as aspects are loaded, the VM internal type and method objects to store the TPTL, per-method cache and other advice related information.

In addition, our implementation counts on the stack layout and calling conventions used by the baseline compiler. Our interaction with the stack layout is simply to copy values farther down the stack. This ensures that we do not need to modify the maps the garbage collector uses to walk the stack.

Chapter 5

Performance quantification and analysis

We quantify 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 seven 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. [13] 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 the micro-benchmarks are based on the same program setup and vary primarily in their use of pointcut and advice functionality. Figures 5.1 and 5.2 show the make up of a typical micro-benchmark. We will mention the differences explaining how a benchmark deviates from the code shown in these figures when discussing the individual benchmarks below. 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

```
public class SuperClass {

  private int dummyCounter = 0;
  public int counter = 0;
  public static boolean testSane;

  public void method1() {
    dummyCounter += 2000000014;
    if (dummyCounter == 1)
      testSane = false;
  }

  public void test(int numRepeats) {
    for (int i = 0;
         i < numRepeats;
         i++) {
      method1();
      ... (20 total calls)
      method1();
    }
  }
}


public class SubClass extends SuperClass {
  // Run quick test if invoking directly.
  public static void main(String args[]) {
      (new SuperClass()).test(100);
  }
}
```

Figure 5.1: Classes used in the benchmarks. Note that the exact class instantiated in SubClass.main changes from SuperClass to SubClass depending on the benchmark.

```
public aspect TestAspect {

    private int counter = 0;

    before(): call(* SuperClass.method1(..)) {
      counter += 2000000014;
      if (counter == 1) SuperClass.testSane = false;
    }
}
```

Figure 5.2: An aspect representative of the ones used in the benchmarks. The benchmarks contain advice with a similar body and differ from each other primarily in the number and kind of advice and pointcuts they use.

counter.[6]

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

[6] These micro-benchmarks are part of a suite we have also used for testing JIT optimized code [14]. The counter increment and test are designed to prevent the optimizing compiler from optimizing away the method bodies, calls, and possibly advice entirely.

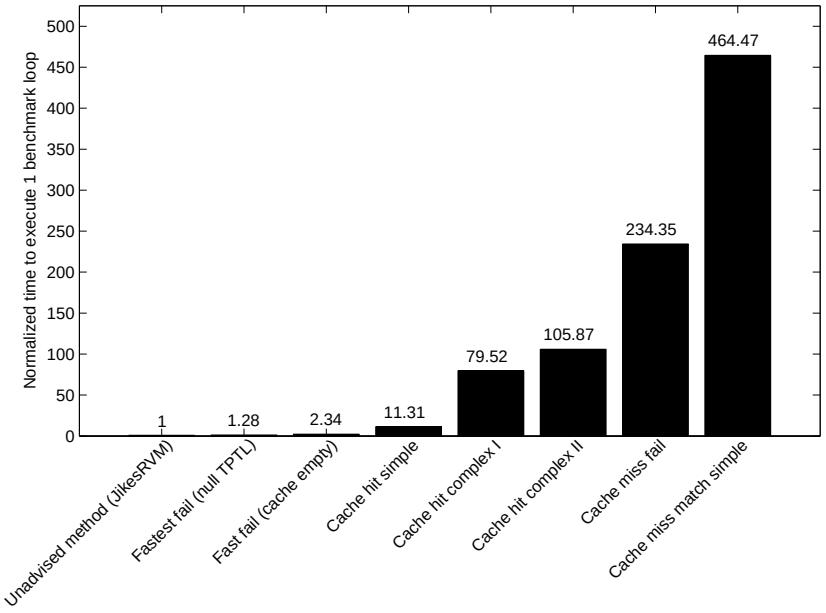


Figure 5.3: .
Quantitative benchmark results normalized to "Unadvised Method (JikesRVM)." Bars are annotated with their value.

version 1.5.2[7] in our benchmarks.

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.[8] We disable the JIT compiler for these benchmarks because we want

[7] We had to modify the ajc load time weaver slightly to stop it from weaving its own classes and going into an infinite loop when run under Jikes RVM version 2.9.2.

[8] Jikes RVM production mode means that many assertion and sanity checks are disabled to improve VM execution speed, and the core VM classes are pre-compiled with the optimizing JIT compiler.

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 5.3 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.

We present the details of these benchmarks below. We also show the advice used by the benchmarks where applicable. Note that unless otherwise indicated, the advice bodies contain the same code as shown for the advice in figure 5.2.

Baseline method call (JikesRVM)

This benchmark measures the baseline cost of an unadvised call to a nearly empty method (the method just includes the counter increment and test). It does not contain any aspect. This benchmark was run on an identically configured plain RVM without our extensions.

Fastest fail (null TPTL)

This benchmark measures the fastest path to failure through the interpreter. This occurs when there is an empty TPTL for a given STT. The `TestAspect` does not contain any advice for this benchmark.

Fast fail (cache empty)

This benchmark 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).

```
before(): call(* SuperClass.noSuchMethod(..)) { ... }
```

Cache hit simple

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

```
before(): call(* SuperClass.method1(..)) { ... }
```

Cache hit complex I

This benchmark 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 an argument type as well as the target type, and an `after` advice which also extracts the argument value from the call.

The `SubClass.main` method instantiates a `SubClass` object for this benchmark so that the `target` pointcut matches at runtime.

```
before(): call(* SuperClass.method1(..)) { ... }

before(String s): call(* SuperClass.method1(..))
                    && args(s)
                    && target(SubClass) { ... }

after(String s): call(* SuperClass.method1(..))
                   && args(s) { ... }
```

Cache hit complex II

This benchmark measures the cost of executing a method call with two applicable advice: a simple `before` advice and an `around` advice that captures a String argument and performs a dynamic type test on the target.

This benchmark contains the same code for `SubClass.main` as the previous benchmark.

```
before(): call(* SuperClass.method1(..)) { ... }

void around(String s): call(* SuperClass.method1(..))
                         && args(s)
                         && target(SubClass) {
  counter += 2000000014;
  if (counter == 1) SuperClass.testSane = false;
  proceed(s);
}
```

Cache miss fail

This benchmark 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 behavior of our implementation. This benchmark uses the same advice as in *Fast fail (cache empty)*.

Cache miss match simple

This benchmark 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. This benchmark uses the same advice as in *Cache hit simple*.

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 5.3 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 as shown in Figure 4.1. 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×) 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 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.

Unadvised DJP execution

This benchmark 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(* AnotherSubClass.method1(..)) { ... }
```

Before call DJP execution

This benchmark 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.

```
before(): call(* SuperClass.method1(..)) { ... }
```

Dynamic state capture

This benchmark 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.

The `SubClass.main` method instantiates a `SubClass` object for this benchmark so that the `args` pointcut matches at runtime. The advice body uses the `SubClass`'s counter instead of the aspect's counter.

```
before(SubClass c): call(* SuperClass.method1(..))
                        && args(c) { ... }
```

Around advice

This benchmark 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.

```
int around(int x): call(* SuperClass.method1(..))
                       && args(x) {
  counter += 2000000014;
  if (counter == 1) SuperClass.testSane = false;
  return proceed(x);
}
```

CFlow w/o and with state capture

These benchmarks defines two advice both 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.

The `SuperClass.test` method for this benchmark calls a method which recursively calls `method1` two times. The advice body of the second advice uses the `SuperClass`'s counter instead of the aspect's counter.

```
before(): call(* SuperClass.method1(..))
    && cflow(execution(* SuperClass.method0(..))) {
            ...
}


before(SuperClass c): call(* SuperClass.method1(..))
          && cflow(execution(* SuperClass.method0(..))
          && this(c)) { ... }
```

Multiple advice

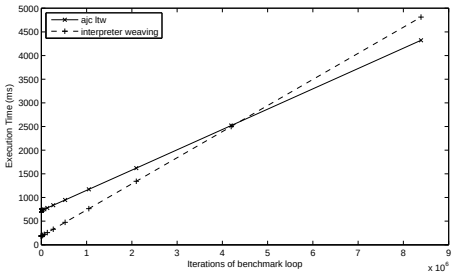
This benchmark involves multiple advice on the same DJP. It is a combination of three of the previous benchmarks. It incorporates the advice from *Dynamic State Capture* and *Around Advice* into one aspect, and has a second aspect which is the same aspect used in the *Before Call DJP Execution* benchmark.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

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.

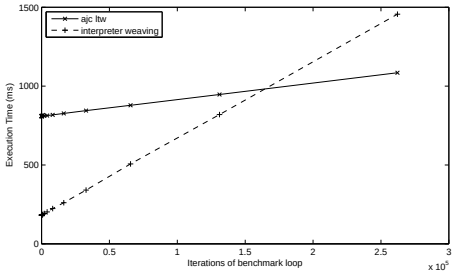
Both VM implementations are built in production mode, but they are configured to disable recompilation by the optimizing JIT compiler. We disable the JIT because we only want to measure the performance of code that has been through the baseline compiler – i.e. what we consider interpreter code in this experiment.

The implication of this for our benchmark results is that 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, the performance curves reported by each benchmark would likely 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.)

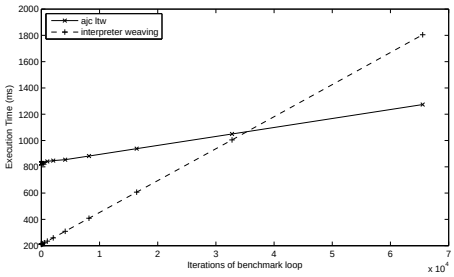
The results of our performance measurements are shown in Figure 5.4. We graph the results with running time in milliseconds along the y-axis, and number of loop iterations along the x-axis. Only iterations of powers of 2 were actually computed, values between these results were linearly interpolated. 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



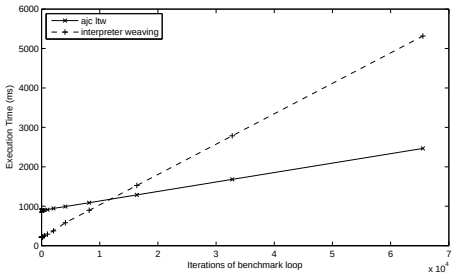
(a) Unadvised DJP Execution (9×10^6)



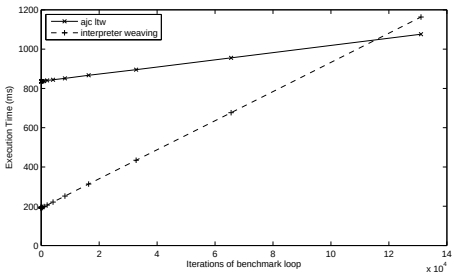
(b) Before Call DJP Execution (3×10^5)



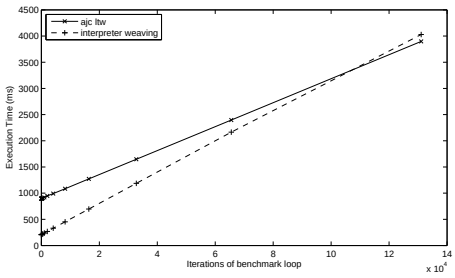
(c) CFlow w/o State Capture (7×10^4)



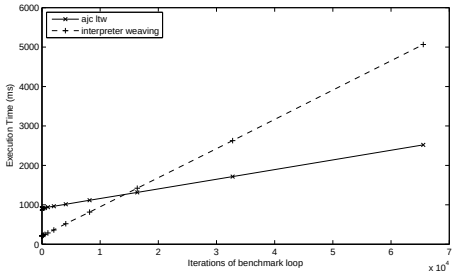
(d) CFlow w/ State Capture (7×10^4)



(e) Dynamic State Capture (14×10^4)



(f) Around Advice (14×10^4)



(g) Multiple Advice on DJP (7×10^4)

Figure 5.4: .
Comparative benchmark results (scale)

Benchmark	to intersection				Slope
	Iterations	Unadvised DJPs	Advised DJPs	Time (ms)	
Unadvised DJP Execution	4,367,900	436,790,000	0	2600	1.2841
Before Call DJP Execution	164,520	23,032,800	3,290,400	980	4.6271
Dynamic State Capture	115,450	16,163,000	2,309,000	1050	4.0488
Around Advice	107,370	15,031,800	2,147,400	3350	1.2725
CFlow w/o State Capture	35,324	4,945,360	706,480	1067	3.5615
CFlow w/ State Capture	11,846	1,658,440	236,920	1177	3.2404
Multiple Advice	14,240	3,702,400	284,800	1262	3.0035

Table 5.1: 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, number of unadvised and advised DJPs executed before intersection, 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.

interpolation of the data yields nearly straight lines.

We report up to 2^{15} iterations for each of the benchmarks except for the Unadvised and Before Call for which we report up to 2^{19} and 2^{17} iterations respectively because the intersection points are further along. Note that for these two benchmarks, the scale of the x-axis differs from the others.

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.

The results of these benchmarks are also summarized in Table 5.1. In this table we report the number of iterations of the benchmark loop required before the curves cross over, and 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 seven 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 5.1 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), or startup, performance. Again, this is not surprising.

It is important to remember, as mentioned in the previous section, that these benchmarks were run without enabling recompilation by the JIT compiler. So, it is not a given, were recompilation enabled, that these performance curves would always intersect in the manner that they do. In fact, they would not intersect at all provided that JIT recompilation occurred before the time saved at startup was equaled by the time lost to the slower interpreter. Note that in even the worst performing pointcut and advice benchmarks, namely "CFlow w/ State", it takes at least 11,000 iterations before the curves cross and our implementation begins performing worse than ajc ltw. Furthermore, the most common case, unadvised DJPs is optimized most dramatically and requires more than 4,000,000 iterations of the loop before the curves intersect.

5.3 Implications for hybrid implementations

Hybrid virtual machines will not improve the performance of all applications all the time. Rather, the purpose of including an interpreter in the VM is to balance the trade-off between interactivity and startup performance on the one hand vs. performance of long running programs on the other hand. The interpreter is intended to execute code that runs infrequently (and just for a short time) reasonably well. The optimizing compiler is called to optimize long or frequently running methods to improve performance. The interpreter must execute well enough that it can execute short running programs faster than compiling and executing the compiled version of those programs. In addition, it should support the optimizing JIT compiler by providing profiling and other runtime data which allows the JIT compiler to run efficiently and make appropriate optimization choices.

With this in mind, several points appear to be significant with respect to a possible hybrid weaving implementation – interpreter based weaving together with JIT time rewrite based weaving.

First, startup times of the interpreter weaving implementation are shorter than for the ajc ltw implementation by 600-700ms across the benchmarks. This translates to the interpreted execution of 100-117 million unadvised DJPs. As discussed in Section 5.2 the per DJP overhead of the interpreter weaving approach vs. ajc ltw ranges from 28% to 462%. Table 2.1 suggests it will be closer to 28% more than 90% of the time, since most DJPs are

unadvised. The comparative benchmarks show that given the interpreter's 'head start' of 600-700ms, roughly 436 million interpreted unadvised DJP executions would need to occur before the ajc ltw implementation would catch up in the common case.

We also note that 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.

Finally, once a method has been JIT compiled, the per-shadow method cache can be garbage collected (or returned to a free list) 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 simply not possible to draw definitive conclusions about the potential benefits of a hybrid approach. But the 600-700ms head start, 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.

Chapter 6

Related work

The related work can be analyzed along several different dimensions. One dimension is whether the system supports static or dynamic aspect deployment. Systems supporting static aspect deployment have the complete list of aspects available at either compile time, post compile time or when the application starts running, whereas systems supporting dynamic aspect deployment have to be able to load new aspects and unload previous ones during an application's execution.

Another dimension is whether the system performs offline or online advice weaving. Offline advice weaving happens either at compile time or post compile time, whereas online advice weaving happens either at class load time, or when the method is being interpreted or JIT compiled.

Offline weaving approaches have to scan all the DJP shadows and then rewrite potentially advised shadows to invoke the advice guarding it with any residual dynamic checks. It leads to the least overhead at runtime as it pays the matching and rewriting cost offline. However, offline weaving is infeasible in some cases, such as when the programmer is going through rapid edit-compile-debug cycles where fast turn around time is more important than execution efficiency and where the cost of scanning all the DJP shadows can be prohibitive. It is impossible in other cases such as when the list of aspects is not available till the application starts running.

Online weaving approaches get to see fewer DJP shadows but they incur more overhead per DJP as they have to perform the matching and code rewriting at runtime. Some approaches perform non-rewriting based weaving saving the overhead of code rewriting but at the cost of reduced execution efficiency. Other approaches, particularly those supporting dynamic aspect deployment, rewrite code to insert hooks at DJP shadows which call back into the VM to check for advice applicability when the DJP runs. This enables them to cheaply support the addition and removal of advice at runtime but at the cost of reduced advice execution efficiency.

While approaches supporting static aspect deployment have mostly focused on offline or load time weaving, and approaches supporting dynamic aspect deployment have mostly focused on online weaving, more overlap is

definitely possible. Our work is an attempt towards supporting a static aspect deployment semantics through a combination of different online weaving approaches. Similarly, while approaches supporting dynamic aspect deployment will always need to support mechanisms to weave and unweave advice at runtime, they can take advantage of offline weaving to weave in aspects known before runtime, emitting information which allows for later unweaving and reweaving as new aspects are added and old ones are removed at runtime.

We now contrast and compare our system with the different systems reported in the literature.

6.1 ajc

The most commonly used AspectJ implementation is ajc [21]. As mentioned earlier, it is a bytecode rewriting based implementation. It has modes which allow it to perform the rewriting based weaving at compile time, post compile time and runtime as the classes are being loaded by the VM. The load time 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 ltw will run the matcher against all shadows that are loaded. This coupled with the fact that we do not perform code rewriting at advised DJPs allows our approach to have better application startup performance but at the cost of reduced advice execution efficiency.

When weaving advice at shadows in a class, ajc's matcher uses a *fast-match* strategy[18] which is used to discard those pointcuts from consideration whose applicability is scoped by a `within` or `withincode` pointcut and which cannot possibly match shadows in the class. Our matching strategy differs from ajc in that we organize the advice by STT type patterns mentioned in the pointcuts. Instead of testing the same set of pointcuts at each DJP shadow in a class, we only test those pointcuts at a shadow that are reachable through the TPTL of the STT of the DJP shadow.

6.2 AspectWerkz

AspectWerkz[9] [8] 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.

6.3 abc

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.

6.4 JAsCo

JAsCo [31, 33, 32] implements support for an 'aspect enabled' Java beans component model. Its implementation 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 at runtime. For optimization purposes, JAsCo also supports the ability to inline the advice dispatch code[32], 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 code rewriting for advice weaving.

[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.

6.5 PROSE

PROSE [29, 28, 27] 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.

The first version of PROSE[29] used the Java VM debugging interface to add hooks into the running program to effect advice dispatch, whereas the second version of PROSE[28] integrated directly with the RVM's baseline compiler. Instead of generating machine code to directly call advice, the second version implemented hook based weaving where machine code was generated at each DJP shadow to call back into the runtime weaver if the DJP shadow was found to be potentially advised at runtime. While our implementation has similarities to that of PROSE, there are important differences. PROSE has to scan all loaded classes to search for potentially advised DJP shadows which it then "activates". The hooks are responsible for calling back into the VM when they reach an activated DJP shadow. In contrast, the only time we ever have to look into a class is when the *.tnp list is not empty, and even then, we lazily scan only those classes that appear as STTs in the DJPs for which the out of line path gets called. Secondly, PROSE generates machine code at each DJP shadow to help enable atomic advice deployment at runtime. We do not have to deal with atomic advice deployment as we only support static aspect deployment.

The third version of PROSE[27] added support for runtime weaving by modifying the VM's internal JBC representation. This differs from our approach in that we do not modify the JBC at all. While the complete hybrid implementation strategy discussed in section 5.3 will perform rewriting based weaving, that rewriting will be integrated with the JIT compilation of the method and will not require modifying the byte code of the method.

6.6 Steamloom

Steamloom [5, 17], 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.

A later version of Steamloom implemented efficient cflow testing inside the VM[6]. Our implementation uses a counter based strategy for cflow testing, similar to that used by ajc[18], but since our implementation is integrated with the VM, it can in principle be extended to support a similar strategy as Steamloom for efficient cflow testing.

6.7 Nu

The Nu [11, 10] 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 [25] 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.

Finally, it is useful to note that the relative performance of the Nu implementation and ours are not directly comparable because of several differences in approach. First, our implementation supports seven DJP kinds in AspectJ's *region-in-time* join point model whereas Nu supports two DJP kinds in the point-in-time model. Furthermore, the correspondence of DJPs between these models is not straightforward: a call in Java gives rise to one DJP in AspectJ, but three DJPs in the point-in-time model. Second, method dispatch in the Hotspot interpreter is about 3.5 times slower than method dispatch in the Jikes RVM baseline compiler, which means that the same absolute difference in overhead per method execution shows up as a significantly different relative performance overhead between the implementations. Finally, the data gathering methodology used by the Java Grande benchmark suite, used in [10], is not statistically rigorous, and instead reports results from one execution.

6.8 ALIA

ALIA [7] 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.

Chapter 7

Future work

7.1 Lookup key

For simplicity, we key advice lookup on a single DJP shadow property – the static type of the target.

One area of future work will be to explore keying advice lookup on different properties for different kinds of DJPs. For example, if target name patterns turn out to be a better indicator of advice applicability as compared to STT type patterns, then we can key advice lookup on target names for DJP shadows that have a target name and key on the STT for the rest of the shadows. Since there are many more names that types, the VM will have to spend more time constructing the data structures and use more space to store them. Thus, while keying on the target name might decrease the number of unadvised DJP shadows at which we have to follow the out of line path, it is not entirely clear whether following such an approach will clearly lead to performance gains.

Another potential optimization will be to have different tables for the different kinds of DJPs instead of dividing a single table into different parts for different kinds of DJPs. This way the absence of advice at DJPs of those kinds could be tested directly from the inline machine code.

7.2 TPTs for the enclosing type's type patterns

One way the AspectJ pointcuts narrow the scope of advice applicability is through the use of `within` and `withincode` pointcuts. In addition to having TPTs for the STT type patterns, we might also have added support for TPTs for the enclosing type's type patterns (thereafter referred to as *within TPTs*) mentioned in the `within` and `withincode` pointcuts. While storing the advice which narrowed their scope through `within` and `withincode` pointcuts in the within TPT instead of the STT TPT would have limited the lookup for that advice to only shadows within the enclosing type, that advice would have had to be looked up for each shadow in the enclosing type. Further-

```
public aspect Aspect {
  before(): call(* C1.m1(..)) &&
              !cflow(within(Aspect)) {
      ...
  }
}
```

Figure 7.1: A before advice with a pointcut which prevents the advice from applying at any DJP in the dynamic extent of the shadows in the aspect.

more, this would have had doubled the amount of code to be executed in the fastest fail path thereby having a large impact on the the overall application performance. We decided not to support within TPTs for these reasons.

However, it must be noted there are some cases where the absence of a within TPT can noticeably slow down the performance of the overall application. Programmers sometime write advice of the form shown in figure 7.1. Since the cflow's inner pointcut does not specify any STT, the counter manipulating advice ends up being stored in the * TPT and thus being checked at each DJP shadow. Having within TPTs in addition to STT TPTs would drastically improve the performance of applications containing such pointcuts. We decided not to implement support for within TPTs because we were not sure of the frequency of such advice in practice and because we did not want to slow down applications which did not use such pointcuts. One area of future work will be to collect data from a larger set of aspectj applications and implement support for within TPTs if such pointcuts are found to be used widely.

7.3 Support for the complete language

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 thesis 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 will have to maintain a separate list of advice that can match at advice execution DJPs and check the advice in that list before invoking any advice. The abstract evaluation rules in figure 3.2 can be modified straightforwardly to determine which advice can match at

advice execution DJPs and thus should be put in that list.

7.4 Shadow cache

We presently allocate a vector of words of size equal to the number of byte-codes plus one for the per-method shadow cache. While this allows for fast testing of the absence of any applicable advice, this uses a lot of space. One area of future work will be to explore the space-time trade-offs associated with using a more compact per-method shadow cache.

We use the cache vector to store several different pieces of information at present. It is be used to test whether the shadow has been seen before, whether it is advised, as well as to get the dispatcher caller if it is advised. One simple optimization will be to store these different pieces of information in different more space efficient data structures. We can store information about whether the shadow has been seen before in an array of bits instead of an array of words. We can use another array of bits to test whether the shadow is advised, and search a linked list of dispatcher callers to find the relevant dispatcher caller if it is advised. Since only a small percentage of shadows are actually advised as shown in table 2.1, we can achieve further space optimizations by using a bloom filter[4] to test whether the shadow is advised instead of an array of bits. Bloom filters are a space efficient data structure used to test whether an element is a member of a set. Members can be added to a bloom filter but they cannot be removed. In addition, they sometimes result in false positives but never in false negatives. Since we support static aspect deployment, we will only need to add elements to the bloom filter and never remove any. Furthermore, even though false positives will result in the execution of the out of line path slowing down the execution of the DJP, they will not lead to incorrect behavior. As such, bloom filters will be adequate for our use but we will have to experiment and collect data in order to determine good values for the different parameters of the bloom filter.

7.5 Generating specialized code for more cases

As described in section 4, we generate specialized dispatchers, dispatcher callers and dynamic check objects for a number of cases. One area of future work will be to collect data from a representative set of aspectj applications to find out the common patterns of advice and pointcuts that programmers use and generate specialized code for any common case that we missed.

We generate specialized dispatcher callers for cases where only a single advice is applicable at a DJP shadow and delegate the rest of the cases to the generic dispatcher caller. We can improve on this by generating specialized dispatcher callers for efficiently calling different ordered pairs of advice at DJP shadows. Since the number of different cases to be implemented increases drastically with the number of advice applicable at a DJP shdow, it will be useful to collect data for deciding which combinations of advice to generate specialized code for.

We generate specialized dynamic check objects only for those dynamic residuals that contain pointcuts strung together only by an && pointcut and fall back to the generic dynamic check object for the other cases. Another area of future work will be to generate specialized dynamic check objects for other simple patterns of residuals, such as residuals containing pointcuts in disjunctive normal form (OR of ANDs), conjunctive normal form (AND of ORs) and other simple patterns.

7.6 Integrated LTW

We made our comparisons against an external load time weaving implementation: ajc ltw. The ajc ltw cannot access the internal representation of types loaded by the VM which leads to redundant loading of types by the ajc ltw. 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 [14, 5], but none of these implementations have been optimized for startup efficiency). One avenue of future work will be to work on an optimized load time rewriting based weaver and compare the performance of our implementation with it.

7.7 Hybrid implementation

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.

Chapter 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 a load time rewriting approach, and the execution efficiency is still good enough that there appears to be sufficient time to optimize the code with the JIT compiler.

Bibliography

[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] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] 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.

[6] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient control flow quantification. *SIGPLAN Not.*, 41(10):125–138, 2006.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] Etienne Gagnon and Laurie Hendren. Effective inline-threaded interpretation of java bytecode using preparation sequences. In *in Compiler Construction, 12th International Conference*, pages 170–184. Springer, 2003.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] Angela Nicoara, Gustavo Alonso, and Timothy Roscoe. Controlled, systematic, and efficient code replacement for running java programs. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 233–246, New York, NY, USA, 2008. ACM.

[28] 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.

[29] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM.

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

[31] 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.

[32] Wim Vanderperren. Optimizing jasco dynamic aop through hotswap and jutta. In *Dynamic Aspects Workshop*, pages 120–134, 2004.

[33] 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.

[34] 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`.