An Execution Layer for Aspect-Oriented Programming Languages

Michael Haupt   Mira Mezini   Christoph Bockisch   Tom Dinkelaker
Michael Eichberg   Michael Krebs
Software Technology Group
Darmstadt University of Technology, Germany

{lastname}@informatik.tu-darmstadt.de, tom.dinkelaker@web.de, krebs-m@gmx.de

ABSTRACT

Language mechanisms deserve language implementation effort. While this maxim has led to sophisticated support for language features specific to object-oriented, functional and logic programming languages, aspect-oriented programming languages are still mostly implemented using postprocessors. The Steamloom virtual machine, based on IBM's Jikes RVM, provides support for aspect-oriented programming at virtual machine level. A bytecode framework called BAT was integrated with the Jikes RVM to replace its bytecode management logic. While preserving the functionality needed by the VM, BAT also allows for querying application code for join point shadows, avoiding redundancy in bytecode representation. Performance measurements show that an AOP-enabled virtual machine like Steamloom does not inflict unnecessary performance penalties on a running application; when it comes to executing AOP-related operations, there even are significant performance gains compared to other approaches.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*run-time environments*

General Terms

Languages, Performance

Keywords

Aspect-oriented programming, virtual machine support

1. INTRODUCTION

This paper is about providing support for aspect-oriented programming (AOP), more specifically for the pointcut-and-advice (PA) flavour of AOP [25], in a Java virtual machine. The fundamental concepts of the PA flavour of AOP are

join points, *pointcuts* and *advice*. A join point is a point in the execution of a program. A pointcut is a query that quantifies over join points, thereby defining related sets of join points. Advice are pieces of functionality that can be attached to pointcuts, taking semantic effect when the respective pointcuts "match", i.e., when join points referred to by the pointcut are reached.

For illustration, consider the AspectJ [24, 3] code in Fig. 1. The class `Display` serves as a canvas for shape objects, such as `Points` and `Lines`. The aspect `DisplayUpdating` implements a protocol between these classes which says that "any time a method is called on a shape object which changes the state of the shape, the display must be updated". The ability to quantify over execution points via the pointcut `change()`, and to express common behavioural effects for all these points in the `after` advice associated with `change()`, makes it possible to express the protocol in a modular unit. Without the aspect, the implementation of the display updating protocol would be scattered over call sites, or the bodies, of such state-changing methods.

The implementation of a PA language consists of two main building blocks: a join point shadow retrieval module (JPRM) and a weaving module (WM). The JPRM maps dynamic join points to their corresponding static shadows [26]: code structures (expressions, statements or blocks) that *might* yield dynamic join points during execution. A method execution join point's shadow is a method body, the shadow of a method call is a call instruction, etc. Given a pointcut, the JPRM calculates the shadows of join points matched by the pointcut. The shadows are passed to the WM; the latter weaves code for dispatching to aspect functionality at these shadows.

For pointcuts that quantify only over static properties of join points, and can thus be directly mapped to code, the dispatching logic is a direct call to advice functionality. However, pointcuts that quantify over dynamic properties of join points such as those that use `cflow`, `target`, `this`, or `args` pointcut designators[1] in AspectJ cannot definitely be mapped to places in code; for such pointcuts the dispatching logic also includes pieces of conditional logic (called *residuals*) to check for the dynamic properties. Depending on the kind of dynamic pointcuts, the implementation of the residuals can be more or less complex.

---

[1]The `cflow` designator quantifies over control flows, and `target`, `this` and `args` can be used to filter objects from a join point's context by type, where the latter applies to, e.g., method parameters.

```
public class Display {
  private Shape[] shapes;
  private Display instance;
  private Display() {...}
  public static Display instance() {...}
  public void addShape(Shape s) {...}
  public void update() {...}
}

public interface Shape {
  public void moveBy(int dx, int dy);
}

public class Point implements Shape {
  private int x,y;

  public void setX(int x) {...}
  public void setY(int y) {...}
  public void moveBy(int dx, int dy) {...}
}

public class Line implements Shape {...}

aspect DisplayUpdating {
  pointcut change():
    call(void Point.set*(int))
    || call(void Shape.moveBy(int, int));

  after(): change() {
    Display.instance().update();
  }
}
```

Figure 1: Display update code.

For `target`, `this`, and `args`, residuals can simply be implemented as dynamic type checks. Residual logic gets more complicated for `cflow`; in this case, the application's execution and its entering and leaving control flows need to be monitored. Recent advances in the development of pointcut languages [27], however, go much further with regard to dynamic properties of join points that pointcuts can refer to, taking into account the *history* of application execution, or the dynamic object store. While such models increase the power of pointcuts as referencing mechanism, improving information hiding, dynamic residual logic gets also more complicated [27]. Finally, there are AOP implementations that support "dynamic weaving" [4, 21, 30]: in these systems, it is possible to weave/unweave aspects into/from a running application. Under such circumstances, the set of join point shadows cannot in all cases be determined statically. Due to this, the aforementioned systems insert additional residuals at any potential join point shadow.

Currently, dispatching logic, including residuals, is inserted into application code at compile- or load-time. Hence, this logic is executed by the VM as part of the application. Language mechanisms are thus implemented at application level, not at language implementation level. This semantic gap has important effects on performance and debugging which will be discussed in detail in Sec. 2 and which have been the motivation for the work presented in this paper.

It is our conviction that aspect-oriented language mechanisms, just like previous paradigms' language mechanisms, such as late binding, lazy evaluation or unification, deserve to be integrated in the underlying execution environments.

Our previous work on Steamloom [8, 16] has indicated that AOP support at VM level can be realised without significant performance overhead on a running application. Moreover, advice dispatching logic can benefit from the VM's replacement of conditional logic with the implicit invocation of advice in only those places where they *have* to be applied, avoiding most of the residuals used otherwise.

However, the version of the VM presented in [8, 16] supports a fairly primitive set of AOP features. The only supported join point type was *method execution*, for which both join point shadow retrieval and weaving are trivial to implement: the join point shadows are simply the first instruction of a method, all returning instructions, and those throwing instructions whose exceptions are not caught within the same method. While this first experiment actually served well the goal of motivating work on VM-level support for AO features, the actual work remained yet to be done.

This is the focus of this paper. The main contribution of the work presented here is the integration of both the JPRM and WM into the VM for supporting AspectJ's dynamic join point model. Technically speaking, the integration is realised by replacing the VM's original management logic for method bytecodes with a new solution provided by the Bytecode Augmentation Toolkit (BAT [6]). BAT provides powerful facilities for querying bytecodes and for inserting and removing instructions into and from existing code. Both the VM core and the AOP logic operate on the very same data structures. This has the effect that, while the AOP functionality can rely on BAT's features to address join point shadow retrieval and weaving, the VM can transparently use the same interface as before to deal with method bytecodes. In essence, Steamloom allows for pointcut evaluation and aspect weaving while the application is executed by the VM. The approach of providing services for join point shadow retrieval at VM level is novel, and Steamloom is the first AOP environment to offer such support.

The paper is structured as follows. In the next section, we will discuss several AOP implementations with respect to AOP infrastructure presence at application level. In Sec. 3, we will briefly describe the Jikes RVM with special regard to the features we have used when implementing Steamloom. Sec. 4 gives a brief overview of Steamloom. In Sec. 5, we discuss the integration of BAT into the Jikes RVM to extend Steamloom's join point and weaving model. Results from performance measurements are presented and discussed in Sec. 6, along with optimisation challenges. Sec. 7 concludes the paper and discusses future work directions.

2. RELATED WORK

There are three kinds of code that we subsume under the term *infrastructural code*. In the previous section, we have described residuals and advice dispatching logic. The third kind of code comprises of all code that forms the actual *infrastructure* of a particular AOP implementation. This includes, for example, classes for the representation and evaluation of pointcuts, for weaving aspect code into the application, for associating aspect instances with application objects, for (de)activating aspects at run-time, and so forth.

For illustration, we have analysed the control flow that is executed when a method `C.m()` is decorated with a simple *before execution* advice using AspectWerkz 1.0. A simplified version of the control flow is displayed in the sequence diagram in Fig. 2. The original method `C.m()`'s body was

replaced with a call into the AspectWerkz infrastructure. In the `BeforeAdviceExecutor` (marked with `(*)` in the figure), a loop iterates over all `before` advice attached to the method's execution. In the loop, the actual advice (the method `Before.before()`) is eventually invoked via reflection. The original implementation of `C.m()` has been moved to a new method whose name is displayed in abbreviated form in the figure, which is also reflectively invoked.

All in all, a significant amount of infrastructural code is executed for a simple call to `C.m()`, including two reflective method invocations. Their cost leads to this solution not scaling too well if, for example, multiple advice are involved in the decoration of a running application. Even if the advice attached to `C.m()` are revoked by undeploying the aspect defining them, the original method is still invoked reflectively after some AOP infrastructure has been executed.

We will now analyse several prominent approaches to AOP with respect to the amount of infrastructural code present at application level. Apart from Steamloom, which will be introduced in Sec. 4, we identify two categories of AOP systems. The classification allows for investigating the various approaches with regard to the way infrastructure is realised in them: the question is whether the AOP infrastructure provided by an implementation is part of the application being executed, or indeed part of the execution layer that runs the application.

In the first category, there are systems that offer AOP support at *application level*: they express their AOP-enabling infrastructure in Java bytecode which is loaded and executed by the respective VM, just like the application itself. AspectJ [24, 3], JBoss AOP [21], and EAOP [12, 13] fall into this category. Among them, EAOP is implemented as a preprocessor, JBoss AOP relies on a modified class loader that performs weaving operations at load-time, while the AspectJ implementation comes in two flavours, based on compiler and, respectively, class loader support. Despite the differences in their implementations, they have in common that their AOP support is implemented at application level and thus subject to execution by the VM.

The second category consists of systems that, while having most of their infrastructure still implemented at application level, use *VM services* to realise dynamic aspect weaving. In some cases, small core portions of AOP support are even implemented as VM extensions. JAsCo [33, 34, 19] and AspectWerkz [4] rely on the Sun VM's HotSwap capabilities [11]. AspectWerkz uses HotSwap to modify methods at load-time, while JAsCo selectively recompiles methods as the application is running.

PROSE [30] also belongs to the second category. Its architecture provides a unified high-level AOP layer called the *dynamic AOP engine* [29] that relies on varying low-level implementations of the *execution monitor* [29]. The execution monitor exists in two versions. The first one relies on the standard JVM's debugger to register join points as breakpoints and intercept execution there [28]. A small core written in C as a VM plug-in provides the necessary adaptation of debugger breakpoints to join points. The second execution monitor implementation [30] is an extension to the Jikes RVM [23]. It basically provides mechanisms for join point notification via callbacks and dynamic method recompilation. The second approach aims at providing the same features as used in the debugger-based implementation while avoiding the severe performance drawbacks (cf. Sec. 6).

To summarise, all of the systems mentioned above expose significant parts of their AOP infrastructure to the VM for execution and thus handle it as part of the application. The question to be raised is to what degree AOP infrastructure should be present at application level. Is it part of the application, or is it part of the underlying execution architecture running the application?

An AOP implementation is meant to work transparently in the sense that an advice invocation should simply happen *implicitly*, instead of leading to the invocation of numerous infrastructural methods that eventually invoke the actual advice. In this regard, we identify several shortcomings of AOP implementation approaches at application level.

The first issue is isolation of infrastructural code from application code. It comes into scope when an application built using AOP features is debugged. All systems offering AOP support at application level (in Java bytecode) suffer from the debugger not being able to tell application code, including aspects, and infrastructural code apart. This also affects profiling in that infrastructural code is profiled as well as the actual application.

The fact that, at least in the dynamic AOP approaches like JBoss AOP, AspectWerkz and PROSE, advice methods are not called directly but from infrastructural code by reflection, even leads to a considerable performance drawback of such approaches. Advice invocations are, in such cases, naturally not as fast as they could be, were they directly woven into affected code. Moreover, reflective invocation of advice prohibits inlining, the most important optimisation to be applied by a JIT compiler.

As we will also show in Sec. 6, the impact of integrating infrastructural code into the application is less problematic in AspectJ as compared to other approaches. The reason is that AspectJ does not support dynamic weaving of aspects—hence, residuals are kept minimal.

To conclude, it is always technically possible to provide AOP support at application level. However, there will always be both conceptual and performance drawbacks, as discussed above. A fully integrated approach like Steamloom, on the other hand, offers great opportunities for implementing dedicated support in the run-time architecture itself, leading to both conceptual and performance advantages. For example, advice instance tables [16] are not feasible at application level.

3. THE JIKES RVM

IBM's Jikes RVM [1, 2, 23] is almost entirely written in Java. The fact that Java disallows some operations such as direct memory or CPU register access is met by the bootstrapping and just-in-time (JIT) compilation technique which compiles Java bytecode to native machine code and applies to each method being executed. When building the RVM, a boot image is created containing the machine code for all classes as well as all objects that constitute the virtual machine. The boot loader is the only part of Jikes that is not written in Java.

The RVM has two different just-in-time compilers (JITC), the first of which, called *baseline compiler*, does not abstract from the stack machine as which a JVM is specified. The second JITC has three levels of optimisation. An example for possible optimisations is inlining, which is the most aggressive and effective among the optimisations. Additionally, Jikes implements an adaptive optimising compiler sys-
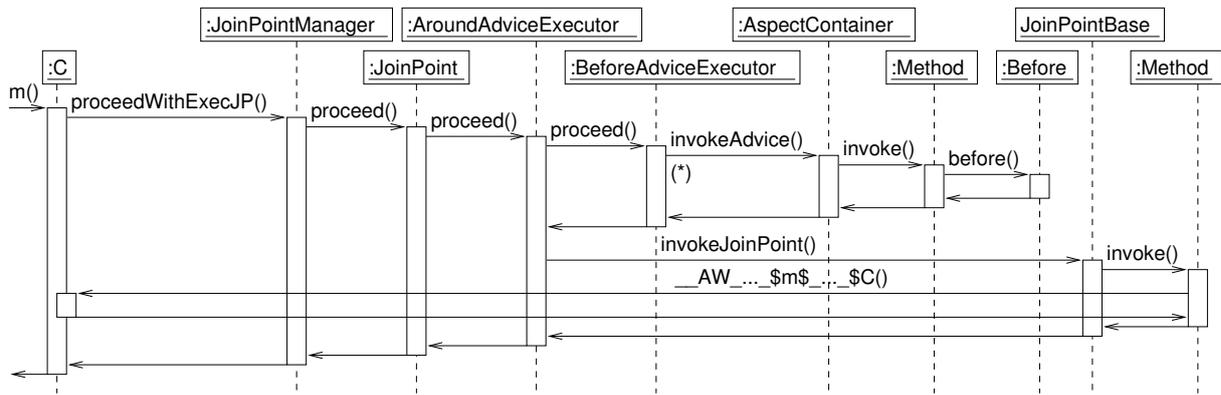
Figure 2: Sequence diagram of a before advice invocation in AspectWerkz 1.0.

tem [10] employing a low overhead sampling mechanism to allow for profile-driven optimising recompilation. Methods can also be recompiled while they are on the call stack [15].

Class loading as implemented by Jikes is almost entirely dynamic. The bootstrapping, however, requires some classes necessary for executing the RVM itself to be in the boot image and thus to be loaded before run-time. These classes are for the most part Jikes' components but also some of the classes from the standard library, such as `String` and `File`, are included in the boot image. Application classes are, in contrast, always loaded dynamically.

When loading a class, its bytecode is parsed and represented by a tree of objects whose root is formed by an instance of `VM_Class`. The leaves are objects for the class' methods (class `VM_Method`) and fields (class `VM_Field`). Via reflection these objects can be accessed, but unlike the standard Java reflection they also provide insight into the internals of loaded classes, their methods and fields.

For each loaded type, including classes and array types, a type information block (TIB) is constructed which contains, among other type-specific properties, the type's virtual method table. An instance of a type consists of several memory slots holding pointers to the dynamic type's TIB, field values or array elements, and other instance-specific properties. So, all instances of a type share the TIB. Static fields as well as static methods are stored in a global table, called Jikes Table of Constants (JTOC).

4. STEAMLOOM IN A NUTSHELL

Steamloom is an extension to the Jikes RVM that provides AOP support for Java applications. The AOP functionality is implemented at virtual machine level and accessible through a Java API. Steamloom's dynamic join point model is as powerful as that of AspectJ.

In Steamloom, an aspect is a mere container that maps pointcuts to advice. Aspects are first-class entities that can be defined, activated and deactivated (deployed and undeployed in Steamloom terminology) while an application is running. Aspect deployment takes place by sending `deploy()` to an aspect instance; aspects are undeployed by calling `undeploy()`.

Steamloom currently supports `before` and `after` advice; `around` advice are being implemented. In Steamloom, advice are arbitrary – static or virtual – methods of arbitrary classes. An instance of the `Advice` class represents a con-

```
public class DisplayUpdating {
  public static Aspect setupAspect() {
    Method updMethod =
      Display.class.getDeclaredMethod("update", null);
    Advice updAdvice =
      new AfterAdvice(updMethod, Display.instance());
    PointcutDesignator change =
      SimpleParser.getPointcut(
          "call(void Point.set*(int)) || " +
          "call(void Shape.moveBy(int,int))"
      );
    Aspect a = new Aspect();
    a.associate(updAdvice, change);
    return a;
  }
}
```

Figure 3: Display updating in Steamloom.

crete advice, encapsulating a reflective representation of the advice method in question.

To provide an impression of the way aspects are dealt with in Steamloom, we will now briefly describe how the display updating example shown in AspectJ syntax in Fig. 1 can be implemented in Steamloom. Fig. 3 contains the code for the entire aspect functionality.

The actual display updating aspect is assembled in and returned from the `setupAspect()` method. The aspect's semantics are the same as for the one presented in Fig. 1. We will now describe the way it is assembled through the Steamloom API in more detail.

The first two lines of the method simply retrieve a Java reflection object representing the method `Display.update()` which is henceforth to be used as advice body. The next two lines create an `Advice` instance. Any such instance comprises a method constituting advice functionality – in this case, `Display.update()` – and an instance to which the advice method invocation is to be sent. In our example, this so-called *advice instance* is the singleton `Display` instance. The advice is an `after` advice.

In the next few lines, a pointcut is assembled. Steamloom's pointcuts are represented as object hierarchies of `PointcutDesignator` instances. While complex pointcuts can be assembled by hand, we have used a parser for pointcuts in AspectJ syntax in this example for convenience. This parser is part of the Steamloom implementation.

In the last three lines of the method, an actual aspect instance is created, and a mapping from the `change` pointcut to the defined advice is defined in the aspect. Finally, the aspect is returned from the method.

It has to be noted that the sole existence of this class does not entail any assumption as to when the aspect defined by `DisplayUpdating` is woven and when it starts to take effect. Since aspects are first-class entities in Steamloom, the `Aspect` instance returned by the `setupAspect()` method in Fig. 3 can exist for some time without affecting the running application in any way. However, as soon as it is sent the `deploy()` message, the aspect will be inserted in the running application and the additional functionality it defines will immediately be attached to all calls to the methods mentioned in the `change` pointcut.

All Steamloom data structures, such as aspects, pointcuts and advice, are inherent parts of the VM. Therefore, a call to an aspect's `deploy()` method is not a simple application method invocation, but a direct call *into* the VM itself. The API is also not necessarily intended for direct use by a programmer, as seen in Fig. 3. It is rather the surface of the VM's built-in support for – theoretically arbitrary – AOP languages and may as well be targeted by a compiler.

5. JOIN POINT SHADOW RETRIEVAL AND WEAVING

In order to integrate AOP support into the VM, we have modified the implementation of the Java language meta-model and the static representation for bytecode instructions of the RVM. Aspect weaving in Steamloom is done by first modifying affected methods' bytecodes and then recompiling them, during which process all optimisations applied upon further compilations are retained. Jikes' on-stack replacement mechanism is also supported. In a nutshell, the entire set of optimisation strategies Jikes provides is available in Steamloom.

More details about the way the AOP infrastructure was integrated into the VM will be presented in the following sub-sections. We illustrate that employing an additional layer of abstraction for bytecode in a VM is practical and requires adaptations in few well localised places throughout the VM. Additionally, we show that the applied modifications do not critically interfere with other subsystems of the VM. Thus, these subsystems are preserved as reusable.

5.1 BAT

In this section, we briefly introduce the bytecode toolkit BAT [6], used by Steamloom for join point shadow retrieval and weaving. BAT offers functionality to change existing bytecode and to create completely new sequences of bytecode instructions. In BAT, all information stored in a class file is made available by a fine-grained object hierarchy, the meta-model entities, down to the level of an instruction, i.e., every bytecode instruction is represented as an object. A method's instruction sequence is stored as a doubly-linked list making updates of bytecode sequences efficient.

An important feature which makes BAT particularly well suited as part of an AOP-enabling infrastructure is its framework for efficient and fine-grained localisation of join point shadows, the so-called *bytecode pointcut framework*. Filter objects can be composed to describe the join point shadows to be selected by their properties. Such properties can be,

for example, various elements of the signature of a method call/execution, or a field access join point shadow's field name, etc. The bytecode instructions that pass the filter are returned when the filter is applied to a class file representation. In addition, there are filters that represent the logical "`&&`", "`||`" and "`!`" operators, which can be used to build more complex filters. For illustration, regard the following filter that selects all instructions that access the field `out` declared in the class `java.lang.System`:

```
Filter f = new AndFilter(
  new FieldAccessNameFilter("out"),
  new FieldAccessDeclaringClassFilter(
    "java.lang.System"
  )
);
```

BAT analyses composite filter objects and optimises them to reduce the number of steps required when applying the filter to a class file.

Unlike other bytecode toolkits [7, 22], BAT declares interfaces for all of the Java language's meta-model entities. In Steamloom, the meta-model classes for provided by the Jikes RVM have been modified to implement these interfaces: that way, BAT can access them when evaluating filters. The possibility of using the adapter pattern to integrate the bytecode framework with the VM is, in addition to the bytecode pointcut framework, a second important reason for the preference of BAT over other toolkits.

Within Steamloom, BAT takes over the complete bytecode instruction management for application classes thereby facilitating retrieval of join point shadows, and enabling to manipulate the bytecode instructions which is not possible in Jikes. Below, we will discuss how BAT is integrated into the Jikes RVM and how Steamloom makes use of it.

5.2 Translating Pointcuts to BAT Filters

Aspects in Steamloom are defined in terms of pointcuts rather than BAT filters. Steamloom supports AspectJ pointcut expressions, which it translates to filters, so that BAT can be used to retrieve the respective join point shadows. To support AspectJ-like pointcuts, a class hierarchy is provided where each pointcut designator is represented by a specific class. As in AspectJ [3], they can be parameterised by patterns, e.g., specifying the set of methods a method call pointcut designator selects. There are also classes in the pointcut designator hierarchy, which represent logical operators, permitting to combine several sub-pointcuts to create complex pointcut trees. For illustration, consider the pointcut `change()` from the example in Fig. 1. Using the aforementioned pointcut designator class hierarchy, this pointcut is represented as shown in Fig. 4. The root of the pointcut tree is an instance of the class `OrDesignator` and the leaves are instances of the class `CallDesignator` which are further parameterised by method patterns.

To use BAT for join point shadow retrieval, Steamloom creates a tree of BAT filter objects out of this pointcut expression. For example, the pointcut represented by the object structure in Fig. 4 is translated into the filter tree shown in Fig. 5. The use of `OrFilter` and `AndFilter` in Fig. 5 is self-explanatory. The other filters restrict the instructions that can be a join point shadow to those `invoke` instructions whose signature, visibility, name, and declaring class matches the given patterns.
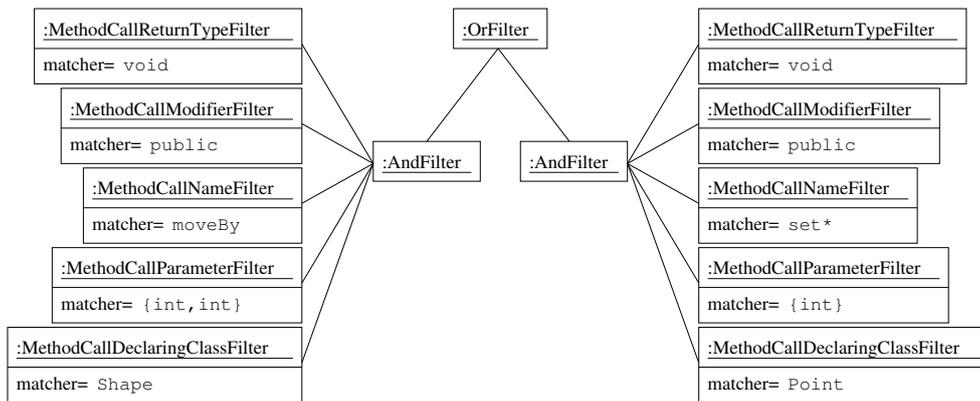
```
:MethodCallReturnTypeFilter          :OrFilter          :MethodCallReturnTypeFilter
matcher= void                                           matcher= void

  :MethodCallModifierFilter                               :MethodCallModifierFilter
  matcher= public                                         matcher= public

    :MethodCallNameFilter     :AndFilter   :AndFilter        :MethodCallNameFilter
    matcher= moveBy                                          matcher= set*

  :MethodCallParameterFilter                              :MethodCallParameterFilter
  matcher= {int,int}                                      matcher= {int}

:MethodCallDeclaringClassFilter                         :MethodCallDeclaringClassFilter
matcher= Shape                                           matcher= Point
```

Figure 5: BAT instruction filter objects, resulting from the pointcut designator structure in Fig. 4.

```
                 call(public void Shape.moveBy(int,int)) ||
                 call(public void Point.set*(int))

call(public void                :OrDesignator          call(public void
Shape.moveBy(int,int))                                  Point.set*(int))

                    :CallDesignator   :CallDesignator

                    :MethodPattern    :MethodPattern

                 public void              public void
                 Shape.moveBy(int,int)    Point.set*(int)
```
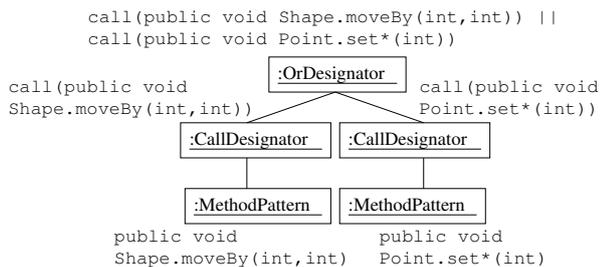
Figure 4: Pointcut designator objects.

When the BAT filter tree in Fig. 5 is evaluated, the result set contains all `invokevirtual` instructions in all application classes where the called method is either one of

```
Shape.moveBy(int, int),
Point.setX(int),
Point.setY(int),
```

or any other method in any subclass of `Point` whose name begins with `set`, or any implementation of `moveBy()` in a subclass of `Shape`.

5.3 Bytecode Instruction Management and Weaving

Let us now discuss how BAT replaces Jikes bytecode instruction management for application classes and how aspect weaving with BAT is realised in Steamloom.

In Jikes, a method's bytecode instructions are represented as `byte` arrays. This is not ideal for AOP support, since weaving frequently encompasses adding instructions to existing code: arrays cannot easily be expanded. BAT uses doubly-linked lists of instruction objects, which is more flexible and allows for manipulating the instructions easily.

We call the process that transforms bytecode instructions from the fixed array representation into the flexible representation *BATification*. The BATification process is carried out at class loading time, but only for application classes; we do not want to allow the VM's functionality or classes from the run-time library to be decorated with aspects. Our opinion is that aspects are only to be used for interfering with the application, while the aforementioned classes are part of the execution environment.

To allow its subsystems, e. g., the JIT compiler, access to bytecode instructions, Jikes provides an abstraction layer, the so-called *bytecode stream*. For methods of system classes whose bytecode instructions are still represented as `byte` arrays, Jikes' original implementation is used, which is basically an iterator on an array of bytes. For BATified methods, Steamloom provides an alternative implementation of the bytecode stream which implements the same interface as the original Jikes bytecode stream but iterates over BAT instruction lists. The differences of the two bytecode stream implementations are hidden by their common interface, thus dependent subsystems of the VM can access method instructions regardless of them being BATified or not.

Once join point shadows for a pointcut are localised, resulting in a set of selected instruction objects, the advice associated with the pointcut at hand is woven before or after the shadow, depending on the advice type. Steamloom does so by using BAT to weave in a call to the method specified as advice.

Code that is woven in is surrounded by two instructions, which are specifically introduced by Steamloom, namely the `beginadvice` and `endadvice` instructions. They contain an identifier of the aspect they belong to and mark the code block containing the advice method call and, if necessary run-time checks, which are explained below. They serve to facilitate undeployment of specific aspects without reweaving all other aspects that affect a method. When an aspect is undeployed, the instructions framed by `beginadvice` and `endadvice` instructions with the respective aspect identifier are simply removed from the instruction list. Both instructions are treated like `NOP` instructions by the compiler (which effectively means they are ignored) and thus introduce no execution overhead. They just serve as tags.

Advice are, as mentioned above, normal Java methods. In case an advice method is virtual, advice invocations are sent to objects, so-called *advice instances*. They are directly associated with the classes they advise through *advice instance tables* [16], an efficient concept for storing advice instances. For their lookup, another specific bytecode instruction was added to the VM that retrieves an advice instance from the table in minimal time.

These special-purpose bytecodes do not have to be generated by any compiler; they are for sole use by the Steamloom infrastructure when it generates dynamically woven code. They are therefore not expected to be present in any

class file that is loaded into the VM. The bytecode verifier may safely treat them as illegal when it comes across them during class loading.

5.4 Non-Statically Determinable Pointcuts

Join point shadow retrieval as described above only applies to static pointcut designators, such as `call` or `get`. Dynamic pointcut designators, such as `this` or `cflow`, only yield *potential* static join point shadows. We will now discuss how these designators are handled in Steamloom.

The `cflow` designator is handled in a special way making use of Steamloom's runtime weaving feature. For illustration, consider a pointcut `pc` defined as `cflow(pc1) && pc2`. For this pointcut, a special advice is woven in before shadows that match the pointcut `pc1`. This special advice weaves in (deploys) the advice originally associated with `pc` at the join point shadows of `pc2`—i. e., the `pc2` shadows are advised only after we have reached `pc1` and, hence, are sure to be in its control flow. After `pc1` shadows, another special advice is inserted that unweaves the original advice. If the control flow described by `pc1` can be entered recursively, Steamloom ensures that the original advice is woven the first time the control flow is entered and unwoven the last time it is left.

The other dynamic pointcut designators are implemented, in a way similar to AspectJ [18]. At all statically determined join point shadows, where the actual advice call depends on the run-time conditions, tests are woven in as well as a conditional branch skipping the advice call if the conditions are not satisfied. For the pointcut

`(pc1 && this(<type>)) || pc2`

the join point shadows determined for `pc2` need no run-time tests. At the join point shadows for `pc1` Steamloom weaves in a test that ensures the dynamic type of the active object is `<type>` when the advice is called.

Steamloom splits pointcuts into sub-pointcuts, so-called *pointcut parts*, whose elements have common dynamic pointcut designators. These parts can be treated separately, where at the join point shadows retrieved for one part, the same run-time tests have to be woven in.

After evaluating all pointcut parts, only the union set of all join point shadows has to be considered for weaving. So, even if multiple pointcut parts specify the same shadows, the advice is only called once. Consider, e. g., the pointcut `call(* *.setX()) || call(* *.set*())`: a call to a method called `setX()` is specified by both pointcut parts.

When building the union set of join point shadows, the runtime tests necessary for a shadow are merged, so that the right tests can be woven in. Consider the pointcut

`(pc && this(<type1>)) || (pc && this(<type2>))`

which is split into two parts both leading to the same potential join point shadows. For the set of shadows computed for the first part, the run-time condition must be satisfied that the active object is of dynamic type `<type1>`. For the shadows in the second set, the condition must be satisfied that the active object is of type `<type2>`. In this example merging the run-time conditions results in a test if the active object is either of type `<type1>` or of type `<type2>`.

6. EVALUATION

In this section, we will present and discuss the results of various performance measurements we have applied to Steamloom and a selection of the AOP implementations presented in Sec. 2. The measurements are split up into three groups. We first directly compare the performance of the unmodified Jikes RVM to that of Steamloom when running standard benchmarks. The second group contains overhead measurements: we have run standard benchmarks on the various systems to measure the impact of running an application in an AOP-enabling environment instead of using a plain standard JVM with no AOP extensions. Measurements in the third group are micro-measurements that explicitly deal with the cost of AOP-related operations in the various systems. Moreover, there are measurements that collect performance data of more complex applications that have been extended with aspects.

The systems taken into account are as follows: AspectJ 1.2, AspectWerkz 1.0, JBoss AOP 1.0, PROSE 1.2.1 in its debugger-based version, Jikes 2.3.1, and Steamloom.[2] All experiments were run on a Dual Xeon workstation (3 GHz per CPU) running Linux 2.4.23 with 2 GB memory. Where we did not run Jikes or Steamloom as VM, we used version 1.4.2_05 of Sun's standard Java VM in client mode. For all virtual machines in both benchmarks, the initial heap size was set to 512 MB, and the maximal heap size to 1,024 MB.

It is technically possible to run, e. g., AspectJ on the Jikes RVM. Nevertheless, we have refrained from performing measurements with AOP implementations on Jikes because the comparison we have intended is one of AOP implementations in their "natural" environment, not one of different approaches on different platforms. It has to be noted as well that Steamloom is the only one among the presented systems which cannot be run on another platform since it *is* one. The systems we have compared Steamloom to all run on implementations of the full JVM standard.

6.1 Performance Impact

We will now briefly discuss the cost of the modifications we have applied to Jikes in order to implement Steamloom. The results obtained by running the SPECjvm98 benchmarks [32] on both an unmodified production build of the Jikes RVM 2.3.1 and on a production build of Steamloom are displayed in Fig. 6.

The first and second bars of each group in the figure represent average results from running each of the benchmarks 20 times. These results show that the overhead incurred by the modifications that were applied to Jikes in implementing Steamloom is, on average, practically not present. This is due to the fact that the boot images of Jikes as well as Steamloom production builds contain a highly optimised version of the respective VM: all modifications are optimised as well. In the long run, this reduces the overhead for the SPECjvm98 benchmarks to zero.

However, an overhead is to be expected since class loading and compiling of methods is more expensive in Steamloom because of maintaining instruction object lists instead of byte arrays. The third and fourth bars in each group in Fig. 6 show the execution times of the *first* runs of each of the SPECjvm98 benchmark applications. In all of the first

---

[2] AspectWerkz 1.0 and JBoss AOP 1.0 were the stable release versions of the respective systems at the time this paper was written. The published VM-integrated version of PROSE [29] is no longer available, and a more recent version of it was not working in our environment. The same holds for JAsCo, which crashed during micro-measurement runs.
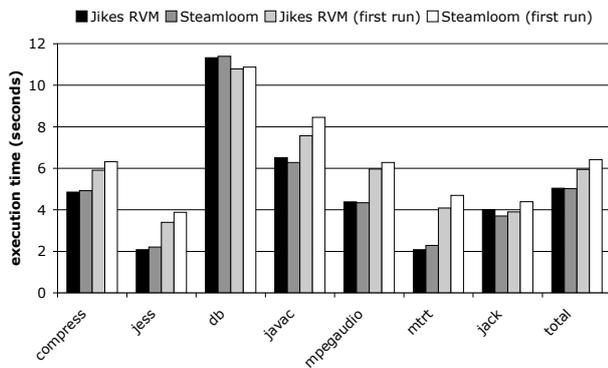
Figure 6: SPECjvm98 measurement results for Jikes and Steamloom.

Figure 7: SPECjvm98 results for all systems.

Figure 8: Results of micro-measurements.

runs, Steamloom performs slower than Jikes, and the average overhead is 7.8 %. This overhead vanishes over larger numbers of benchmark executions because class loading occurs only early during run-time, namely during the first run, and methods are much more often called than compiled.

From these observations, we can conclude that Steamloom brings about the same performance as Jikes; there is no general overhead in execution speed.

6.2 Overhead Measurements

To measure the performance of the various approaches when standard applications are run, the SPECjvm98 benchmark [32] was used. The results were obtained by running each of the benchmarks 20 times at maximum problem size (100) and are shown in Fig. 7. They are averages over all benchmarks. The performance relations between Jikes and Steamloom have already been discussed in Sec. 6.1, where we concluded that using Steamloom does not inflict a performance penalty. On average, the various other systems taken into account that run on the Sun standard VM also bear no significant overhead compared to the bare VM. The worse performance of the two debugger-based systems, AspectWerkz and PROSE, is due to a peak in the `jack` benchmark. For the other benchmarks, these two systems do not exhibit performance penalties.

Fig. 7 also reflects on the performance relationship between the standard VM and Jikes/Steamloom. In most of the benchmarks, Jikes performs better than the Sun VM. This is probably due to Jikes' JIT compilation approach, in contrast to the Sun VM initially interpreting bytecodes.

From the results presented so far, it can be seen that the presence of an AOP infrastructure alone does not inflict performance penalties on running applications that do not employ any AOP functionality. This holds for all categories of AOP implementations. Next, we will consider the cost to be paid for using AOP features in the different systems.

6.3 Cost of AOP Operations

We have measured the cost of AOP operations in the various systems using several approaches, ranging from low-level micro-measurements over a simple application to a single benchmark application from the SPECjvm98 suite.

To measure solely the cost of basic AOP operations, we have used a suite of micro-measurement applications that was developed specifically for this purpose [17]. It is based
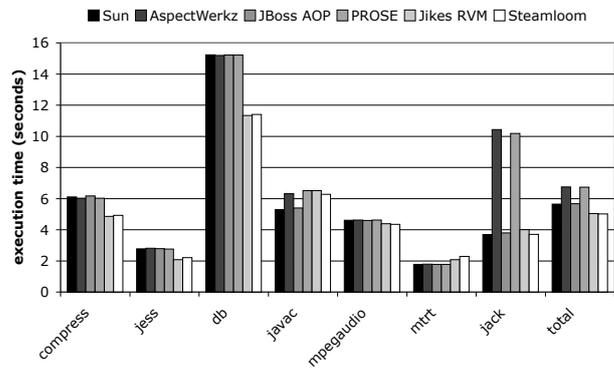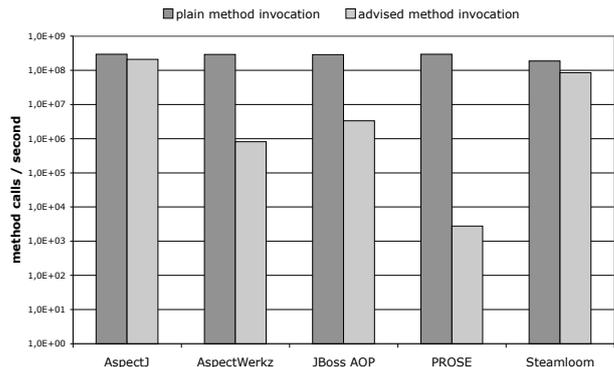
on the JavaGrande framework [9, 20], and results are expressed in operations per second. The specific measurements we have used here determine the cost of virtual method invocations with no aspect functionality (plain invocation), and with a before advice attached to the method execution. Target methods and advice bodies just increment a counter, so that basically only the cost of looking up the advice instance and invoking the advice thereon are measured.

The results are presented in Fig. 8 (the y axis is logarithmic). The first bar in each group denotes the number of calls to a method each of the systems performs when the call is not advised by any aspect. The second bar states the number of method calls advised by a `before` advice (`around` advice in the case of JBoss AOP, since this system only has support for `around` advice).

The impact of running a simple application with aspects decorating it was measured using a program that computes Fibonacci numbers. An aspect was used to count the number of invocations of the recursive `fib()` method. To that end, the aspect attached a `before` advice to the execution of `fib()`. Measurement results were obtained by letting the application compute $fib(20)$ ten times and determining the average time spent for completing that task. In Fig. 9, results gathered from four different configurations of this application are shown. It was first run as a "plain" application, with no aspect activated, but using one of the AOP environments in focus. In the "count all" configuration, the aspect was used to count every execution of the `fib()` method.

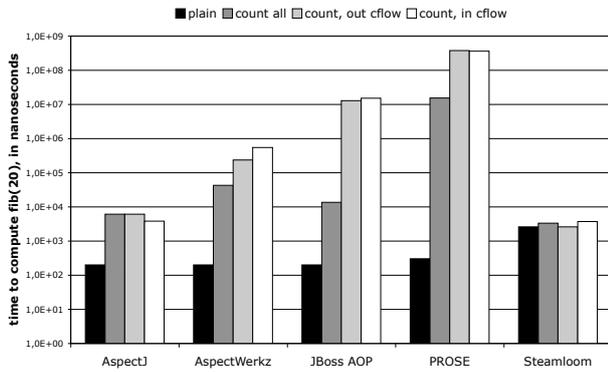When `cflow` is employed, the set of join point shadows cannot clearly be determined statically, so residuals have to

Figure 9: Results of the Fibonacci application measurements.

Figure 10: Results of the SPEC db benchmark with counted method executions.

be used in approaches that use compile- or load-time weaving. While measurements presented so far have only taken into account aspects that do not, when woven into an application, generate residuals, the last two configurations of the Fibonacci measurement deal with the `cflow` construct. Executions of `fib()` were to be counted *only* if the starting call of `fib(20)` originated in the control flow of the execution of a specified method. In the third configuration, this was not the case, while it was in the fourth.

Lastly, a rather complex application, namely the *db* benchmark from the SPECjvm98 suite, was used as the basis of measurements with an aspect with a mostly unanticipated set of join point shadows. The method execution counting aspect from the previous example was used once again, but the set of methods whose executions it had to count was not defined until the benchmark had been started. Thus, compile- and load-time weaving approaches had to introduce a large set of residuals at virtually every method execution (for practical reasons, the set of methods whose executions could be counted was restricted to those methods in the benchmark package).

AspectJ and AspectWerkz do not support late application of advice to join point shadows, so residuals were implemented in aspect code in these cases. JBoss AOP allows for just "preparing" join point shadows at load-time and attaching advice to them later on. PROSE allows for filtering based on regular expressions. Steamloom is the only system in focus that allows for dedicated dynamic decoration of join point shadows with advice invocations.

Once more, the application was executed in "plain" mode to give a reference value. Next, it was run in two configurations with respect to the number of method executions to be decorated. We chose the two possible extremes for this: either *no* method execution was to be counted, or *all* of them. Fig. 10 contains the results from running the SPEC *db* benchmark ten times on each system and computing the average execution time. PROSE, in the latter case, took over 1,100 seconds for a single run, so we have cut the respective bar in the figure to make the others better visible.

In accordance with our observation that employing a runtime environment with AOP support alone does not inflict a performance penalty on a running application, all systems perform comparably well in the various "plain" cases. For calls to advised methods, effects of the different implementation approaches become visible.
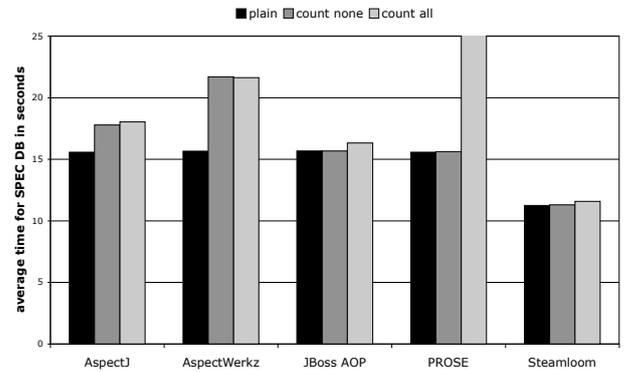
Inserting wrappers at load-time that call advice functionality through a complex infrastructure at application level, as AspectWerkz and JBoss AOP do, clearly has some impact. Profiling the micro-measurements application for AspectWerkz has shown that roughly 26 % of the time were spent in methods responsible for reifying an execution join point or proceeding at one. Profiling JBoss AOP yielded similar results.

The amount of infrastructural code takes even more significant effect when residuals are involved, e. g., when `cflow` is used. While the two systems with optimised support for `cflow`, AspectJ and Steamloom, exhibit no overhead as compared to the non-`cflow` case, the three other systems suffer from their approach: all employ expensive checking mechanisms for testing whether a join point is in a specific control flow. JBoss AOP, for example, does so by creating `Throwable` instances and retrieving stack traces from them.

The residuals that were used in the SPEC db experiment to decorate a given set of method executions with advice also have a performance impact. It is largest in AspectJ and AspectWerkz, where the check for applicability actually had to be implemented as a `HashSet` lookup in the advice code. PROSE is a special case; it suffers from its use of regular expressions, matching whom is time-consuming. JBoss AOP and Steamloom benefit from their minimalistic approaches. JBoss AOP checks for advice applicability at join point shadows with a simple request that is sent to a container object and does not execute anything more when the test fails. Steamloom explicitly only decorates the affected methods.

PROSE, while it also implements significant parts of its infrastructure at application level, suffers additionally from its use of the debugger. As seen in the previous section, running an application in a VM with its debugger activated does, on average, not impose a large overhead on it. Expensive context switches at debugger breakpoints however preponderate when breakpoints are used to reify join points. PROSE can be considered a performant system for AOP applications that do not execute advice too often. In fact, PROSE is intended for use in distributed environments where the performance bottleneck usually is network latency.

On the contrary, both AspectJ and Steamloom perform very well due to their lack of infrastructural code at application level: both systems simply insert direct calls to advice functionality at join point shadows.

6.4 Reflections on Performance Evaluation and Future Optimisations

Observations we have made in developing Steamloom and during performance measurements suggest some optimisations we will perform in the future.

First, recompilation of methods imposes a certain overhead on the running VM: when aspect weaving requires a method to be *immediately* recompiled, the application has to wait for the compilation to finish. The cost introduced by this could be reduced by introducing a *caching* policy for compiled code, so that, whenever a method's code returns to a state it has previously been in, this state is simply restored from the cache, instead of being costly established by recompilation.

Next, the way Steamloom deals with `cflow` at the moment is not optimal in some situations. Recall the idiom `cflow(pc1) && pc2` discussed in Sec. 6.3. In case the control flow of `pc1` is often entered and left, frequent recompilation events occur, leading to considerable performance overheads. Thus, in case `pc1` is very often entered and left, the "classic" approach using a counter to monitor control flow entries and exits as used by AspectJ [26] and the `abc` compiler [5] is more efficient. This approach however suffers when `pc2` is very often matched (cf. Sec. 6.3).

We envision an *adaptive hybrid weaving* approach to meet the requirements of both cases. In that approach, weaving for `cflow` pointcuts follows either the "classic" or Steamloom's continuous weaving strategy based on decisions drawn by profiling data, which the VM gathers anyway to drive adaptive optimised compilation. Apart from that, weaving can anyway be sped up by exploiting native code caching as suggested above. Altogether, we expect this approach to yield better performance in all cases due to its adaptive optimisation characteristics.

Native code caching as suggested above could improve the performance of this approach. However, one might argue that in this case the "classic" approach of monitoring execution as it is used in AspectJ could be applied, but this approach is inefficient in case `pc2` is very frequently matched: every match leads to a checking operation. So, in a nutshell, there are cases when it seems appropriate to deal with `cflow` "traditionally", and there are cases when it is better to follow the continuous weaving approach, as Steamloom does at the moment. A heuristic approach could be used to analyse this based on profiling data, which the VM gathers anyway, and a hybrid weaving approach could be used to apply the appropriate weaving strategy for `cflow`.

Lastly, BAT's representation of bytecode instructions as linked lists of instruction objects, while being ideal for inserting and removing code, is not optimal with respect to memory consumption. It is a strain on the VM's object heap, which becomes apparent when evaluating runs of the SPECjbb2000 benchmark [31] on Jikes and on Steamloom. SPECjbb2000 simulates a multi-threaded client-server application creating large amounts of objects during execution. Results from running the benchmark are shown in Fig. 11 Steamloom, for the standard heap settings used in the measurements, performs clearly worse than Jikes. Due to the large number of small instruction objects, garbage collection reduces the overall throughput of the benchmark. We have also run both Jikes and Steamloom with a larger heap (2,048 MB maximum), and for those runs both VMs exhibit about the same throughput.
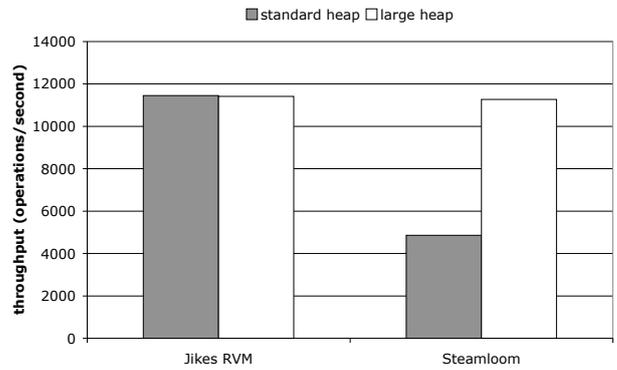

Figure 11: SPECjbb2000 measurement results for Jikes and Steamloom.

The use of BAT was a matter of convenience in this first prototype due to its extended capabilities. A more optimised representation of bytecode that is less explicit in terms of granularity is planned for the future while reusing the approach followed for the integration.

7. CONCLUSION

We have presented Steamloom, an extension to the Jikes RVM that provides VM-integrated support for AOP. To that end, the bytecode management for application classes was entirely replaced by the BAT bytecode toolkit. BAT not only allows for representing Java bytecodes in a way that makes them easily modifiable, but also offers an efficient filter hierarchy to retrieve join point shadows in application code. Steamloom thus is the first AOP environment to offer join point shadow retrieval at VM level.

The fact that BAT is integrated with Jikes to *replace* its original bytecode management logic avoids redundancy in method representation inside the VM. Other approaches that perform dynamic weaving at bytecode level have to work on a duplication of the VM's original representation of a method when applying modifications to it.

Employing an alternative representation of method instructions in a Java virtual machine has proved practical and feasible, and it has required adaptations only in few well localised places throughout the VM. Additionally, the applied modifications of the original VM do not critically interfere with other subsystems thereof.

Performance measurements applied to the Steamloom VM and other AOP systems have shown that using an AOP-enabled run-time environment does not in itself mean that execution is slowed down. However, AOP-related functionality is more efficiently realisable at VM level.

Some of the future work directions are obvious, such as the implementation of missing features like around advice. Also, the optimisations outlined in Sec. 6.4 will be applied. In addition, there are some other more exciting and challenging potential extensions that we plan to work on in the future.

The integration of querying capabilities at the VM level is a good starting point for implementing more expressive pointcut languages based on richer dynamic semantic information. While being the model in use today, the application's static code structure alone – currently represented by BAT objects – is not the most appropriate model for dynamic properties of programs and does not have to remain

the only program model over which queries can be executed. Other models, such as the application's control or data flow history, are conceivable and highly valuable in increasing modularity by enabling pointcuts that directly refer to semantics of join points to be selected [27]. VM-integrated solutions like Steamloom could be crucial in efficiently implementing such dynamic pointcuts by making direct use of VM facilities that collect data on an application's run-time behaviour, such as profiling data which is otherwise only exploited by an optimising just-in-time compiler.

8. ACKNOWLEDGEMENTS

9. REFERENCES

[1] B. Alpern, D. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*. ACM Press, 1999.

[2] B. Alpern and et al. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1):211–238, February 2000.

[3] AspectJ Home Page. `http://www.eclipse.org/aspectj/`.

[4] AspectWerkz Home Page. `http://aspectwerkz.codehaus.org/`.

[5] P. Avgustinov et al. abc: An Extensible AspectJ Compiler. In *Proc. AOSD 2005*. ACM Press, 2005. to appear.

[6] BAT Home Page. `http://www.st.informatik. tu-darmstadt.de/pages/projects/BAT/`.

[7] Byte Code Engineering Library (BCEL) Manual. `http://jakarta.apache.org/bcel/manual.html`.

[8] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proc. AOSD 2004*. ACM Press, 2004.

[9] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000.

[10] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Java Grande 1999 Proceedings*, pages 129–141. ACM Press, 1999.

[11] M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Workshop on Engineering Complex Object-Oriented Systems for Evolution, Proceedings (at OOPSLA 2001)*, 2001.

[12] R. Douence and Mario Südholt. A Model and a Tool for Event-Based Aspect-Oriented Programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

[13] EAOP Home Page. `http://www.emn.fr/x-info/eaop/`.

[14] R. E. Filman, M. Haupt, K. Mehner, and M. Mezini (eds.). Proceedings of the 2003 Dynamic Aspects Workshop. Technical Report RIACS Technical Report No. 04.01, RIACS, 2004.

[15] S. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. `http://www.research.ibm.com/ people/s/sfink/papers/cgo03.ps.gz`, 2003.

[16] M. Haupt and M. Mezini. Virtual Machine Support for Aspects with Advice Instance Tables. First French Workshop on Aspect-Oriented Programming (JFDLPA), Paris, France, Sep. 14th, 2004. `http://www.st.informatik.tu-darmstadt.de/ database/publications/data/JFDLPA04.pdf?id=102`.

[17] M. Haupt and M. Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. In M. Weske and P. Liggesmeyer, editors, *Proc. Net.ObjectDays 2004*, volume 3263 of *LNCS*. Springer, 2004.

[18] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *Proc. AOSD 2004*. ACM Press, 2004.

[19] JAsCo Home Page. `http://ssel.vub.ac.be/jasco/`.

[20] JavaGrande Benchmarks Home Page. `http://www.dhpc.adelaide.edu.au/projects/ javagrande/benchmarks/`.

[21] JBoss AOP Home Page. `http://www.jboss.org/ developers/projects/jboss/aop.jsp`.

[22] Jikes Bytecode Toolkit Home Page. `http://www.alphaworks.ibm.com/tech/jikesbt/`.

[23] The Jikes Research Virtual Machine. `http: //www-124.ibm.com/developerworks/oss/jikesrvm/`.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. Lindskov Knudsen, editor, *Proc. ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[25] H. Masuhara and G. Kiczales. A Modeling Framework for Aspect-Oriented Mechanisms. In *Proc. ECOOP 2003*, 2003.

[26] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In G. Hedin, editor, *Proc. CC 2003*, volume 2622 of *LNCS*, pages 46–60. Springer, 2003.

[27] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *Proc. ECOOP 2005*, 2005. to appear.

[28] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect-Oriented Programming. In G. Kiczales, editor, *Proc. AOSD 2002*. ACM Press, 2002.

[29] A. Popovici, T. Gross, and G. Alonso. Just-in-Time Aspects. In *Proc. AOSD 2003*. ACM Press, 2003.

[30] PROSE Home Page. `http://ikplab11.inf.ethz.ch:9000/prose/`.

[31] SPECjbb2000 Home Page. `http://www.specbench.org/osg/jbb2000/`.

[32] SPECjvm98 Home Page. `http://www.spec.org/osg/jvm98/`.

[33] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented Approach Tailored for Component Based Software Development. In *Proc. AOSD 2003*, pages 21–29, 2003.

[34] W. Vanderperren and D. Suvee. Optimizing JAsCo Dynamic AOP through HotSwap and Jutta. In [14].