# Optimizing JAsCo dynamic AOP through HotSwap and Jutta

Wim Vanderperren
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 62
wvdperre@vub.ac.be

Davy Suvée
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 65
dsuvee@vub.ac.be

## ABSTRACT
The main drawback of all dynamic AOP technologies available today is the rather high performance overhead in comparison to static weaving approaches. In this paper, we propose an approach to improve the performance of both the interception mechanism and the aspect interpreter of a dynamic AOP system. The interception of the base application is optimized by employing the Java HotSwap technology in such a way that only those joinpoints where aspects are applied upon are trapped. When new aspects are added, all corresponding joinpoints are hotswapped for a trapped version. Likewise, when aspects are removed, the corresponding traps are removed, if no other aspect is applicable at the given trap. In order to improve the aspect interpreter, we propose the Jutta system that allows generating and caching a highly optimized code fragment for each joinpoint. This code fragment contains the combined aspectual behavior for the joinpoint at hand. We integrate HotSwap and Jutta in the JAsCo dynamic AOP system and perform extensive benchmarks to evaluate the performance gain of this approach. In addition, the enhanced JAsCo performance is compared to a selection of current state-of-the-art dynamic AOP approaches. These benchmarks indicate that JAsCo, enhanced with HotSwap and Jutta, is able to improve on the current state-of-the-art performance-wise.

## 1. INTRODUCTION
AspectJ is undoubtly one of the most well-known and mature aspect-oriented approaches available today [1]. AspectJ employs static weaving in order to combine the base program and the aspects. As such, aspects cannot be added or removed at run-time; the application needs to be stopped, compiled and restarted in order to change the aspectual behavior. Aspects however often represent concerns that have to be enabled, altered and disabled quite frequently. Typical examples of such crosscutting concerns are debugging concerns such as logging [12] and contract verification [22], security concerns [23] such as confidentiality and access control, management concerns [24] such as accounting and billing, and business rules [5,15] that describe business-specific logic.

During the last years, a wealth of approaches have been proposed to increase the dynamicity of aspect-oriented programming. Examples include PROSE1&2 [18,17], WOOL [19], JAC [16], EAOP [6], OIF [7], AspectWerkz [3], JBoss/AOP [8], HandiWrap [2], AspectS [9], Caeser [14] and JAsCo [21]. The main drawback of most of these approaches is the rather high performance overhead required for applying aspects dynamically in comparison to statically weaved languages like AspectJ. This overhead stems from 1) the interception system employed to interfere with the regular application execution and 2) the aspect interpreter that evaluates which aspects are available at a certain joinpoint and which executes the appropriate advices. In this paper, we investigate how these two mechanisms can be optimized. In order to improve the interception system, we propose to employ the novel Java HotSwap technology that allows replacing the byte code of a class at run-time. In order to improve the second phase, namely the aspect interpretation part, we propose a generic dynamic AOP optimizer, named Jutta. Jutta enables to generate highly optimized code fragments that contain the combined aspectual behavior for each joinpoint. As a proof of concept, we integrate Jutta and HotSwap in the JAsCo dynamic aspect-oriented programming language.

The next section introduces the JAsCo aspect-oriented approach and elucidates the dynamic AOP features offered by this approach. Section 3 presents the Jutta approach and section 4 illustrates JAsCo HotSwap. In section 5, a detailed performance evaluation is performed that compares the enhanced JAsCo implementation with the original JAsCo implementation and a selection of current state-of-the-art dynamic AOP approaches. Finally, section 6 discusses related work and section 7 states our conclusions.

## 2. INTRODUCTION TO JASCO
JAsCo is a dynamic AOP approach originally aimed at combining ideas of aspect-oriented and component-based software engineering. The next sections shortly present the JAsCo approach and discuss the main dynamic features of JAsCo. For more detailed information about JAsCo, the interested reader is referred to [21].

### 2.1 JAsCo language
JAsCo is mainly based upon two existing approaches: AspectJ and Aspectual Components [13]. The JAsCo language is an aspect-oriented extension of Java that stays

as close as possible to the original Java syntax and concepts and introduces two additional entities: *aspect beans* and *connectors*.

An aspect bean is an extended version of a regular Java bean and is specified independent of concrete component types and APIs, making it highly reusable. An aspect bean contains one or more logically related hooks that describe the crosscutting behavior itself. Hooks are able to define three types of advice, namely before, replace and after, which are equivalent to the before, around and after advices known from AspectJ. Figure 1 illustrates an aspect bean that captures a caching concern. The crosscutting behavior, namely intercepting the invocation and returning a cached result instead, is captured in the `CacheControl` hook. Also notice the special constructor of a hook, which specifies a kind of abstract pointcut (line 8 till 10).

```
1  class CachingManager {
2    Cache cache = new Cache();
3    void setRecyclingRate(int sec) {
4      cache.recylingRate(sec);
5    }
6
7    hook CacheControl {
8     CacheControl(method(..args)) {
9       execute(method);
10     }
11
12     replace() {
13       if(cache.isCached(method,args) {
14         return cache.getCached(method,args);
15       }
16       else {
17         Object result = method(method,args);
18         cache.cache(method,args,result);
19         return result;
20       }
21     }
22   }
23 }
```

**Figure 1: A JAsCo aspect bean for caching.**

A connector on the other hand, is used for deploying one or more aspect beans within a concrete component context As such, a connector allows to explicitly instantiate and initialize hooks. In addition, connectors are able to specify explicit precedence and combination strategies in order to manage the cooperation among several aspects that are applicable onto the same joinpoint. Figure 2 shows a connector that instantiates the `CacheControl` hook of Figure 1 onto the `getHotels` method of a `BookHotel` component.

```
static connector CachingConnector {

  CachingManager.CacheControl ca =
    new CachingManager.CacheControl (
      List BookHotel.getHotels(String)
  );

  ca.setRecylingRate(60);
  ca.replace();
}
```

**Figure 2: A JAsCo connector deploying the caching aspect bean of Figure 1.**

## 2.2 JAsCo technology

In order to implement the JAsCo language, we propose a new component model where traps that enable aspect interaction are already built-in. Ideally, new components are shipped employing this new component model. This way, attaching and removing aspects to components implemented in the new component model does not require any adaptation whatsoever to the target beans. Of course, expecting all components to be developed using this new component model is rather utopic. Therefore, it is also possible to automatically transform a regular Java bean into a JAsCo bean by employing a preprocessor that inserts the traps using byte-code adaptations.

Each trap refers to the JAsCo run-time infrastructure that manages the registered connectors and aspect beans. Figure 3 illustrates the run-time infrastructure schematically. The central connector registry serves as the main addressing point for all JAsCo entities and contains a registry of connectors and instantiated hooks. The connector registry is notified when a trap is reached or when a connector is loaded. As such, the database of registered connectors and hooks is updated dynamically. The left-hand side of Figure 3 shows the JAsCo bean *comp1.* All methods of *comp1* are equipped with traps. As a result, whenever a method is called, its execution is deferred to the connector registry. The main method of communication of Java Beans is event posting, so firing an event also reschedules execution to the connector registry. When a trap is reached, the connector registry looks up all connectors that registered for that particular method or event. The connector on its turn dispatches to the hooks that have been instantiated with the corresponding method or event.

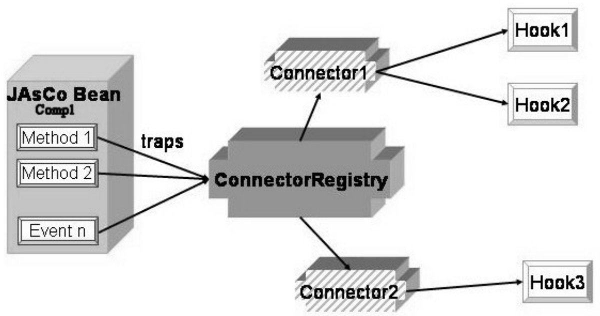

**Figure 3: JAsCo run-time architecture.**

The main advantage of this trapped component model consists of the portability of the approach. JAsCo does not depend on a specialized virtual machine nor on some custom interfaces only available at certain systems. For example, a run-time environment optimized for embedded systems (JAsCoME) and an implementation of JAsCo for the .NET platform have been recently proposed [25]. The drawback is of course that a performance overhead is experienced for all these traps, even if no aspects are applied.

## 2.3 JAsCo dynamic AOP

JAsCo is a dynamic aspect-oriented approach, meaning that new connectors can be added dynamically and obsolete connectors can be removed. When adding or removing a connector, all instantiated hooks are added or removed. JAsCo is one of the most dynamic approaches currently available and offers the following features:

### *2.3.1 Central connector registry*

JAsCo employs a central connector registry that contains all connectors and aspect beans. Without such a central registry, dynamically adding or removing aspects is not flexible at all as one has to iterate over all applicable object instances.

### *2.3.2 Remotely adding/removing aspects*

JAsCo includes a very easy system for remotely (from outside the application) adding a connector. At regular time intervals, JAsCo scans the classpath[1] for new connectors. When a new connector has been found, it is automatically loaded in the system. As such, activating a connector in an application simply means placing the connector in the classpath of the application. Likewise, the removal of a connector is detected by the JAsCo run-time infrastructure and the connector and its instantiated aspect hooks are automatically removed from the system.

### *2.3.3 Precedence Strategies*

Connectors are able to specify precedence strategies that define the priority between advices. In addition, the precedence is able to vary for the different advice types. Figure 4 illustrates a connector that instantiates two hooks: `logger` and `lock`. For the before advices, the behavior of `lock` needs to be triggered first, while for the replace advices, `logger` needs to be triggered first.

```
connector Precedence {
  // hook instantiations...

  lock.before();
  logger.before();
  logger.replace();
  lock.replace ();
}
```

**Figure 4: Precedence strategy in a connector.**

### *2.3.4 Combination Strategies*

Precedence strategies are a solution to some feature interaction problems [26], however other combinations of aspects require a more expressive way of declaring how they cooperate. Therefore, extensible combination strategies are introduced. Combination strategies are implemented using regular Java and are instantiated in a connector. They are able to filter the list of applicable hooks of this connector on a per joinpoint basis. In addition, combination strategies are able to alter the priority and properties of the applicable hooks. Combination strategies are invoked for each execution of the applicable joinpoints. As such, they are able to dynamically influence the combined aspectual behavior. Suppose that an e-commerce system contains two discount aspects, a Birthday discount and a Frequent Customer Discount. Both discounts can however not be accumulated. A combination strategy is able to specify such behavior by removing one of the two discounts when they are both applicable for the joinpoint at hand. Figure 5 illustrates the instantiation of this exclusion combination strategy in a connector and the combination strategy itself.

```
connector DiscountConnector {
  Discounts.Birthday birthday = new ...
  Discounts.Frequent frequent = new ...

  addCombinationStrategy(new
    ExcludeCombinationStrategy(birthday,
      frequent));
}
```

```
class ExcludeCombinationStrategy implements
  CombinationStrategy {
  private Object hookA, hookB;
  ExcludeCombinationStrategy(Object a,Object b) {
    hookA = a; hookB = b;
  }
  HookList validateCombinations(Hooklist list) {
   if (list.contains(hookA)) {
     list.remove(hookB);
    }
    return list;
  }
}
```

**Figure 5: An exclusion combination strategy.**

### *2.3.5 Applying aspects on instances*

It is possible to attach hooks onto specific object instances only, instead of all instances of a particular component type. The concrete instances that are subject of aspect application can be dynamically altered by employing the connector API.

### *2.3.6 Dynamic wildcard matching*

JAsCo also supports the instantiation of a hook on expressions that contain wildcards. These limited regular expressions are matched at run-time. Consequently, when a new component is added to an application, it is automatically affected by all aspects that are instantiated using wildcards.

## 3. JUTTA

## 3.1 Motivation

All dynamic features offered by JAsCo however induce a substantial run-time overhead. The overhead of JAsCo in real-life applications is often more than 1000% in comparison to hard-coding the advices, which is unacceptable. Notice that JAsCo is still in a prototype phase and therefore not a lot of attention has been paid to performance optimizations. The high overhead is mainly

[1] It is also possible to specify a connector loadpath where JAsCo has to search for connectors.

caused by the fact that the entire JAsCo run-time infrastructure is an aspect interpreter. For each joinpoint, JAsCo evaluates which hooks are applicable. When no connectors are added or removed, the set of applicable hooks remains unchanged for every joinpoint. As such, when the same joinpoint is encountered several times, the same logic for finding the appropriate hooks and executing their behavior is computed over and over again. Therefore, a huge performance gain can be realized when the combined aspectual behavior could somehow be compiled and cached for joinpoints that are encountered often. Of course, this compilation process requires some time, but when a joinpoint is encountered a lot, this pays off. In fact, this strategy is similar to just-in-time compilers used in modern virtual machines and therefore our approach is named *Jutta* (Just-in-time combined aspect compilation).

## 3.2 Jutta basics

The Jutta system allows generating and caching a highly optimized code fragment for a given joinpoint. This code fragment directly executes the appropriate advices on the applicable hooks in the sequence defined in the connector. As such, the system avoids iterating over all connectors and its hooks in order to find out which aspectual behavior is applicable. Rearranging the sequence of all applicable hooks for different advice types in order to implement precedence strategies is also avoided. Figure 6 illustrates the simplified Java counterpart of an example cached joinpoint behavior execution. The code fragment first initializes all applicable hooks with the current joinpoint and then executes only those advices that are defined in the connector in the correct sequence.

```
public void executeJoinpoint(Joinpoint jp) {
  hook0._Jasco_initialize(jp);
  hook1._Jasco_initialize(jp);
  hook2._Jasco_initialize(jp);
  hook1.before();
  hook2.before();
  hook0.replace();
}
```

**Figure 6: Simplified Java counterpart of the cached combined aspectual behavior at a joinpoint.**

The current implementation employs the Javassist [4] byte-code manipulation library in order to generate a combined hook behavior code fragment. Using Javassist, a java byte code class representation is generated on the fly, without requiring a compilation step. The overhead of generating a combined hook behavior code fragment is around 10ms on out test system[2]. The optimized code fragment is however only generated when the joinpoint is encountered the first time. As such, for joinpoints that are not executed, no overhead is experienced. The Jutta system also stores all code fragments generated for a given hook combination. As such, when the same hook combination is applicable to a different joinpoint, the overhead for generating the combined hook behavior code fragment is avoided. In addition, the Jutta system includes a set of pre-defined typical combined aspect behaviors. For those combined aspectual behaviors, the generation overhead is also avoided.

The JAsCo approach is however a dynamic AOP approach. As such, the cached behavior for a given joinpoint might become invalid. This happens when a connector is added that instantiates a hook that is applicable on the joinpoint or when a connector is removed that contains an applicable hook for the joinpoint. In addition, it is possible to change some properties of a connector dynamically so that the applicable context of the instantiated hooks is altered. The Jutta system has to be able to cope with these issues.

## 3.3 Hooks depending on dynamic values

Caching combined aspect behavior is not always achievable because it is possible that whether a hook is applicable or not, has to be re-evaluated for every execution of a given joinpoint. For example, when a hook defines a *cflow* condition in its constructor, this constructor has to be re-evaluated for every execution of a joinpoint. However, the entire constructor does not have to be re-evaluated. In this case, only the result of the *cflow* condition is able to change for different executions of the joinpoint. As such, partial evaluation techniques can be used to cache a partially evaluated constructor. In addition, for the particular *cflow* construct, it is sometimes possible to statically analyze whether the condition might ever be true or not by examining the call graph of an application. This technique is elucidated in [20].

## 3.4 Combination strategies

In general, caching the result of the combined behavior of all combination strategies for a given joinpoint is not possible. A combination strategy might depend on dynamic values in order to compute the list of applicable hooks. As such, combination strategies have to be recomputed for each execution of a given joinpoint. Some combination strategies do however not depend on dynamic values and always render the same result for a given input set of hooks. As such, these combination strategies do not need to be recomputed for every execution of a joinpoint. There is however no way to automatically find out whether a combination strategy depends on dynamic values or not. Therefore, the empty interface *DoNotCache* is introduced. When a combination strategy does not implement this interface, it is defined to always return the same set of hooks in the same sequence for a given input set of hooks. As such, the combination strategy only needs to be executed once for every input set of hooks. When a combination strategy does implement the *DoNotCache* interface, it is never cached and thus always executed for each applicable joinpoint.

[2] Pentium4 2GHz, 256MB RAM, Mandrake Linux 9.2, Java 1.4.2

## 4. JASCO HOTSWAP

The Jutta system allows optimizing the aspect interpretation part of JAsCo dynamic AOP. The interception part however is still very slow. Inserting traps at all methods causes a performance overhead for all those methods, even no aspects are applied. In order to optimize this interception system, we propose to employ the HotSwap technology of Java. HotSwap is introduced since Java 1.4 and allows to dynamically replace the byte code of a loaded class. As such, it is possible to install traps just-in-time when a new aspect is added to the system.

### 4.1 Approach

The JAsCo hotswap implementation allows installing traps in only those methods that are subject to aspect application. When a new aspect is added, all the methods where the added aspect is applied upon, are hotswapped at run-time with a trapped version. Because HotSwap does not allow to replace single methods, the complete class byte code is replaced with a version where the applicable methods are trapped. All other methods of the class however remain untouched. Likewise, the original method byte code is installed when the aspect is removed again and if no other aspect is applicable at the method at hand.

The JAsCo HotSwap system does not exclude the regular preprocessing approach for installing traps. Classes that are already equipped with traps using the preprocessor are never altered. As such, when certain classes are definitely affected by aspects, they can be preprocessed to avoid the hotswap overhead at run-time. Furthermore, on platforms where no hotswap virtual machine is available, the preprocessing approach can still be used. JAsCo thus combines the best of both worlds, highly portable through the preprocessing approach and very little overhead when HotSwap is available.

The main drawback of the HotSwap system is that the virtual machine needs to run in debugging mode. As such, a global overhead is experienced depending on the virtual machine implementation. With the introduction of full speed debugging by the newest Sun virtual machines, this overhead is neglectable. However, it appears that on our current Linux virtual machine (Sun JDK 1.4.2_03), a substantial overhead for debugging is still experienced, whereas on the same virtual machine for Windows practically no difference is noticeable.

### 4.2 Implementation Issues

Implementing a HotSwap system for AOP is technically quite challenging. The first problem is that typical HotSwap implementations do not allow altering anything of a class besides the method bodies. In order to implement an efficient AOP system, several fields containing reflective data about the joinpoints contained in the class are however required. Therefore, a separate class containing all those fields is generated each time new traps are installed. As such, the JAsCo HotSwap implementation requires somewhat more memory at run-time than when traps are installed using the traditional preprocesser.

Another problem is that HotSwap only allows replacing classes that are already loaded. As such, when new classes are loaded, the JAsCo run-time infrastructure needs to be notified in order to insert traps at those methods where aspects are applied. The obvious way to realize this is by employing the Java Debugging Interface (JDI) as the virtual machine is already running in debugging mode anyway. Using JDI, an event is received each time a new class is loaded and JAsCo is able to add traps to the methods of this class if necessary. However, by merely setting this "class prepared" breakpoint, the complete application is slowed down by up to 40%! In order to avoid this overhead, another solution is required to receive class loading events. Therefore, JAsCo employs the Java HotSwap facility to hotswap the system class loader by an enhanced class loader that notifies the JAsCo run-time infrastructure whenever a new class is loaded. As such, the overhead for the "class prepared" breakpoint is avoided. Hotswapping the class loader can however cause problems when dedicated class loaders are employed. Typical J2EE application servers [10] depend on a custom class loader system and interfering with this system might cause the application to fail. Therefore, the current JAsCo implementation offers both class loading interception strategies.

## 5. PERFORMANCE EVALUATION

In order to evaluate the performance of the JAsCo approach enhanced with HotSwap and Jutta, we compare it to several state-of-the-art dynamic AOP approaches. Apart from JAsCo, the following dynamic AOP approaches are tested and compared: JBoss/AOP [8], PROSE [18], JAC [16] and AspectWerkz [3]. Notice that this selection is not meant as a comprehensive overview of all existing dynamic AOP systems. We merely selected those systems because they are publically available and seemed stable enough in our opinion. Nevertheless, this selection is a good overview of current dynamic AOP approaches.

We employ two benchmark applications: a benchmark shipped with the JAC distribution and the PacoSuite benchmark [27]. The JAC benchmark application is a synthetic benchmark that invokes a set of public methods with different method signatures and empty method body implementations. This benchmark allows to precisely measure the overhead per method execution for applying aspects. The PacoSuite benchmark is meant as an evaluation of the performance in a realistic and non-trivial application. PacoSuite is a visual component composition environment, which is composed out of 1202 classes containing 34465 lines of code. The PacoSuite benchmark reads an XML composition description from file, validates the composition using a set of finite automata algorithms and finally displays the composition.

The next section shortly discusses the ideas and underlying implementations of the AOP approaches that are used in our experiments. Afterwards, section 5.2 discusses the benchmark results when no aspects are applied. Finally, section 5.3 presents the benchmark results when aspects are applied.

## 5.1 Employed dynamic AOP approaches

When comparing their underlying implementation, JAC and JBoss/AOP are rather similar AOP-technologies. Both approaches make use of traps which are automatically inserted at load-time of the application making use of byte-code transformations. Although both AOP-technologies are quit similar, JBoss /AOP is primarily intended as an aspect-oriented extension for the JBOSS J2EE application server [11], whereas JAC is developed as an AOP-framework which can be used as an alternative for a J2EE application server.

AspectWerkz is meant as a lightweight dynamic AOP framework and also inserts traps at load-time. In addition to employing a customized classloader like JBoss/AOP and JAC, AspectWerkz allows to employ the Java HotSwap functionality in order to hotwsap the system classloader for a classloader that inserts traps. All three approaches however insert traps at load-time. JAC always inserts traps at all methods, while JBoss/AOP and AspectWerkz do only insert traps at classes where aspects are already applied. As such, these approaches are not very dynamic because aspects can only be inserted and removed at trapped methods. Luckily, JBoss/AOP allows specifying a range of classes that have to be trapped, regardless of whether there are aspects applied or not. Unfortunately, for AspectWerkz, this is not possible.

PROSE employs a very different approach to intercept the program's execution than the previous technologies. PROSE exploits the Java Virtual Machine Debugging Interface (JVMDI). A dedicated execution monitor is deployed on top of the JVMDI, which allows capturing relevant execution events. Whenever an event is encountered where an aspect is applied upon, the corresponding advice is executed.

## 5.2 Benchmarks without aspects

For the first experiment, both the JAC and PacoSuite benchmark application are run on our test system[3] making use of the AOP technologies mentioned above, but without any aspects being applied. This allows observing the pure overhead of running the benchmark applications making use the AOP technologies. Notice that both benchmarks first run their application a couple of times in order to allow the Java virtual machine to optimize the code. This also allows some of the approaches to install their corresponding traps (JAC, JBOSS/AOP, JAsCo and AspectWerkz) or to perform some additional optimalisations themselves. The execution times of these warm-up runs are not considered in our benchmark results. For each AOP approach, the experiments are performed at least ten times such that the standard deviation was less than 1% for the JAC benchmark application and less than 5% for the PacoSuite benchmark application.

**Table 1: Benchmarks without any aspects applied.**

| *Without Aspects* | *JAC benchmark* | *PacoSuite benchmark* |
|---|---|---|
| No AOP/AspectJ | 14 ms | 590 ms |
| JAsCo 0.4.5 | 14 ms | 684 ms |
| JAC 0.11 | 154689 ms | - |
| PROSE 1.1.2 | 14 ms | 708 ms |
| JBOSS/AOP 4.0 | 507 ms | 657 ms |
| AspectWerkz 0.9 RC1 | 2651 ms | - |

Table 1 illustrates the result of the first experiment. For the JAC benchmark, one million "direct" iterations are performed. For the PacoSuite benchmark, three "visual" iterations are executed. For JBoss/AOP and AspectWerkz, we made sure that traps are inserted at all methods because otherwise, they are not dynamic at all, as no aspects can be added onto methods that are not trapped. When no traps are inserted, the performance of JBoss/AOP and AspectWerkz is the same as the original application. As explained before, AspectWerkz does not allow specifying that traps have to be inserted, even if there are no aspects. Therefore, we apply an empty aspect to all methods and remove it before the benchmark starts. This way only the overhead of the traps remains. Because removing aspects in AspectWerkz means fetching all possible joinpoints by name, this is not straightforwardly achievable for the PacoSuite benchmark (1202 classes).

At first glance, JAC appears to have a rather big overhead for its own benchmark in comparison to the other AOP approaches. Its low performance is however mainly caused by the slowness of the Java Reflective API which is employed within the JAC implementation. Unfortunately, no JAC results are available for the PacoSuite benchmark, as we were not able to run this application correctly because of JAC code generation errors. Both PROSE and JAsCo perform best in the JAC benchmark as they do not require traps for every method. For the PacoSuite benchmark, the overhead of employing the debugging interface seems to be higher than the overhead of inserting traps at all methods, since JBoss/AOP outperforms PROSE and JAsCo. As already mentioned in section 4.1, this is probably due to less optimal debugging interface implementation on Linux. Nevertheless, inserting traps at

[3]Pentium4 2 GHz, 256MB RAM, Mandrake Linux 9.2, Java 1.4.2

all methods is a feasible approach as the performance overhead in a realistic application scenario is only around 10%.

## 5.3 Benchmarks with aspects

As a second experiment, one simple aspect is applied upon each public method defined within the JAC and PacoSuite benchmark application. This aspect describes an around advice that increases a counter each time it is executed. For this experiment, 100000 "direct" iterations are performed for the JAC benchmark and three "visual" iterations for the PacoSuite benchmark. This results in respectively 800000 encountered joinpoints for the JAC benchmark and 210400 encountered joinpoints for the PacoSuite benchmark application. AspectJ is employed as utopic performance reference. In addition, the performance of JAsCo without the Jutta system being activated is measured. Table 2 illustrates the results. Notice that for the JAC benchmark, only one tenth of the iterations are performed in comparison to the previous experiment (100000 vs 1000000), so the timings for the JAC benchmark of Table 1 and Table 2 cannot be directly compared.

**Table 2: Benchmarks with one around advice applied.**

| *One Around Aspect on all public methods* | *JAC benchmark (800 000 joint points)* | *PacoSuite bench (210 400 joint points)* |
|---|---|---|
| AspectJ 1.1 | 29 ms | 645 ms |
| JAsCo 0.4.5; no Jutta | 424928 ms | 473665 ms |
| JAsCo 0.4.5 | 279 ms | 753 ms |
| JAC 0.11 | 17198 ms | - |
| PROSE 1.1.2 | 946112 ms[4] | - |
| JBOSS/AOP 4.0 | 956 ms | 949 ms[5] |
| AspectWerkz 0.9 RC1 | 487 ms | 3698 ms[5] |

In this experiment, JAsCo clearly outperforms the other approaches. This is mainly the contribution of the Jutta system, which is able to cache the application of aspects such that this information does not need to be calculated each time a joint point is encountered. If the Jutta system is disabled, the performance of JAsCo is very slow and is easily outperformed by all other AOP approaches. Again we observe that JAC, and this time also PROSE, have a rather big overhead in comparison to the other AOP approaches. For PROSE, this big overhead can probably be contributed to the lack of an efficient implementation which is able to cache which aspects are applied on which specific joint points.

In a third experiment, three around aspects are applied upon each public method defined within the JAC bench. As the JAC benchmark application contains 8 public methods, 24 aspect instances are active in the system at the same time. This experiment is mainly performed because caching combined aspect executions is one of the main strengths of the Jutta approach. Table 3 displays the results of this experiment. JAsCo again outperforms the other tested dynamic AOP approaches. However, it seems that JBoss/AOP scales better because adding 23 aspects only increases the execution time for JBoss/AOP with 12% whereas for JAsCo a 35% performance hit is experienced.

**Table 3: Benchmarks with three around aspects.**

| *Three Around Aspects on all public methods* | *JAC benchmark (800 000 joint points)* | *Overhead per advice execution.* |
|---|---|---|
| AspectJ 1.1 | 93 ms | 0.032 ns |
| JasCo 0.4.5 | 395 ms | 0.159 ns |
| JBOSS/AOP 4.0 | 1075 ms | 0.442 ns |
| AspectWerkz 0.9 RC1 | 927 ms | 0.380 ns |

In order to assess the performance gain of the JAsCo HotSwap implementation, a last experiment is conducted. One single around aspect is applied upon one specific method defined within the JAC benchmark application. In addition, each method of the JAC bench is made advisable for JBoss/AOP and AspectWerkz such that aspects can be added at run-time. Notice that this is not required for JAsCo as JAsCo is still able to insert traps in these methods using HotSwap. As illustrated by Table 4, the JAsCo HotSwap implementation improves greatly over the other dynamic AOP approaches as traps are only added at one of the eight methods. Also notice that even the optimized JAsCo system is more than 1000% slower that AspectJ. As such, dynamic AOP is still far behind statically weaved approaches performance-wise.

**Table 4: Benchmark with one around aspect applied upon one specific method.**

| *One Around Aspect* | *JAC benchmark (100 000 joint points)* |
|---|---|
| AspectJ 1.1 | 2 ms |
| JAsCo 0.4.5 | 29 ms |
| JBOSS/AOP 4.0 | 891 ms |
| AspectWerkz 0.9 RC1 | 275 ms |

[4] PROSE does not support an around advice, so we employ a before advice instead.

[5] The actual results for JBOSS/AOP and AspectWerkz are 1093 ms and 4859 ms. Both approaches however also trap private methods. This leads to a higher performance overhead Therefore, the performance of public methods is computed from the overhead per around execution times the number of public methods. Notice that this is not an issue in the JAC benchmark because it only consists of public methods.

As a final note, it should be mentioned that the last three experiments employ an aspect which is described making use of an around advice, as this is the only kind of advice that is supported by each AOP approach that was used within this performance assessment, except for PROSE. Similar to AspectJ however, JAsCo also provides an explicit before and after advice. Apart from the conceptual benefit of an explicit before/after construct, such an advice can be executed faster, as no around advice chain needs to be built up. In case of experiment two for instance, the performance of JAsCo for the JAC-benchmark application is improved by 15% if before advices are applied instead of around advices.

## 6. RELATED WORK

Apart form the approaches employed in our benchmarks, several other AOP approaches are introduced for enabling dynamic AOP. Event based aspect oriented programming (EAOP) allows specifying crosscutting concerns by employing event patterns which are described using a formal language [6]. Because of this formal model, advanced detection and resolution of aspect interactions becomes possible. On the implementation level, EAOP inserts traps that query a central execution monitor, similar to the JAsCo connector registry. The execution monitor has a global view of the executing application and contains all active EAOP artifacts. In contrast to JAsCo, EAOP inserts traps by source-code transformations.

Using Caesar [14], an aspect is described in terms of an Aspect Collaboration Interface (ACI). Each concrete aspect needs to implement the required methods specified by its corresponding ACI. Aspect bindings connect the aspect implementations to different concrete deployment contexts. One of the major contributions of the Caesar approach is the introduction of aspectual polymorphism. Aspect bindings are able to implement a binding for different types and the concrete binding is resolved dynamically using the type of the object at hand. In this viewpoint, aspectual polymorphism is similar to the concept of late binding found in object oriented languages.

Filman [7] proposes dynamic injectors in order to introduce aspects within an application. These dynamic injectors are incorporated into the OIF (Object Infrastructure Framework), a CORBA centered aspect-oriented system for distributed applications. Dynamic injectors are first class objects that can be added and adapted at run-time. At the implementation level, a wrapping approach is employed for injecting the logic of an aspect within a component communication channel.

Wool [19] is a dynamic AOP framework that supports two different dynamic weaving strategies. The Wool system employs the Java Debugging Interface to intercept the execution of the base program. In this respect, Wool is similar to the PROSE approach. However, aspects can also be inserted into the target joinpoints directly by employing Java HotSwap. The original contribution of Wool is that aspects are able to implement their own heuristics for deciding whether they are invasively inserted or not. The difference with the JAsCo hotswap implementation is that JAsCo only insert traps, not full advices. In Wool however, aspects lose their identity at run-time. In addition, Wool requires to hotswap more as for each additional aspect, the classes containing the applicable joinpoints need to be hotswapped again.

Finally, AspectS [9] introduces dynamic AOP support within the Squeak/Smalltalk environment. Pointcuts and their corresponding advices are described making use of plain Smalltalk. By sending the install and uninstall message to an instance of such an aspect, aspects are activated and deactivated within the application at run-time. At the implementation level, AspectS makes use of the dynamic properties of Smalltalk itself. In this case, Method wrappers are used which are placed around a compiled method by replacing its entry in the method dictionary of a class. This way, it is possible to easily add behavior, in this case aspect advices, to method invocations.

## 7. CONCLUSIONS AND FUTURE WORK

This paper presents the HotSwap and Jutta systems in order to improve the performance of JAsCo dynamic AOP. The performance evaluation clearly indicates that the JAsCo implementation enhanced with Jutta and HotSwap improves current state-of-the-art dynamic AOP. However, the overhead is still a lot larger than when statically weaved languages like AspectJ are employed.

The Jutta system is not only applicable to JAsCo. The ideas can be recuperated in any other dynamic AOP approach regardless of which technology is used for intercepting the program execution. Therefore, we plan to decouple the Jutta system from JAsCo and as such achieve a general dynamic AOP optimizer.

The HotSwap system is however only a short and medium term solution for intercepting the program execution. In the long term, the best approach to support dynamic AOP or even regular AOP consists of dedicated aspect-oriented virtual machines as for example proposed by PROSE2 [17]. Indeed, preprocessing, load-time trap insertion or employing the debugging interface of a virtual machine are all solutions that are feasible on the short-term, but are quite cumbersome and error-prone in comparison with a dedicated execution environment. The Jutta system and ideas are however still applicable for such dedicated virtual machines.

## 8. REFERENCES

[1] AspectJ Website. http://www.aspectj.org

[2] Baker, J. and Hsieh, W. *Runtime aspect weaving through metaprogramming.* In Proceedings of the first International Conference on Aspect-Oriented Software Development. Enschede, The Netherlands, April 2002.

[3] Bonér, J. and Vasseur A. *AspectWerkz: a dynamic, lightweight and high-performant AOP/AOSD framework for Java.* Available at: http://aspectwerkz.codehaus.org

[4] Chiba, S. and Nishizawa, M. *An Easy-to-Use Toolkit for Efficient Java Bytecode Translators.* In Proceedings of the second International Conference on Generative Programming and Component Engineering. Erfurt, Germany, September 2003.

[5] Cibran, M., D'Hondt, M. and Jonckers, V. *Aspect-Oriented Programming for Connecting Business Rules.* In Proceedings of the 6th International Conference on Business Information Systems. Colorado Springs, USA, June 2003.

[6] Douence, R., Motelet, O. and Südholt, M. *A formal definition of crosscuts.* In Proceedings of the 3rd International Conference on Reflection. Kyoto, Japan, September 2001.

[7] Filman, R.E. *Applying aspect-oriented programming to intelligent systems.* Position paper at the ECOOP 2000 workshop on Aspects and Dimensions of Concerns. Cannes, France, June 2000.

[8] Fleury, M and Reverbel, F. *The JBoss Extensible Server.* In Proceedings of Middleware 2003 Int Conference, Rio de Janeiro, Brazil, LNCS(2672), January 2003.

[9] Hirschfeld, R. *AspectS – Aspect-Oriented Programming with Squeak.* Objects, Components, Architectures, Services, and Applications for a Networked World, pp. 216-232, LNCS 2591, Springer, 2003.

[10] Java J2EE website. http://java.sun.com/j2ee/

[11] JBOSS J2EE Application Server Website. http://www.jboss.org/

[12] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin J. *Aspect-oriented programming.* In Proceedings of European Conference for Object-Oriented Programming. Jyväskylä, Finland, June 1997.

[13] Lieberherr, K., Lorenz, D. And Mezini, M. *Programming with Aspectual Components.* Technical Report, NU-CSS-99-01, March 1999. Available at: http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html.

[14] Mezini, M. and Ostermann, K. *Conquering Aspects with Caesar.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[15] Ossher, H. and P. Tarr, *Using multidimensional separation of concerns to (re)shape evolving software*. Communications of the ACM 44 (2001), pp. 43–50.

[16] Pawlak, R., Seinturier, L., Duchien, L. and Florin, G. *JAC: A flexible solution for aspect-oriented programming in Java.* In Proceedings of the third International Conference on Reflection. Kyoto, Japan, September 2001.

[17] Popovici, A., Alonso, G. and Gross, T. *Just-in-time aspects: efficient dynamic weaving for Java.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[18] Popovici, A., Gross, T. and Alonso, G. *Dynamic Weaving for Aspect-Oriented Programming.* In Proceedings of the 1st International Conference on Aspect-Oriented Software Development. Enschede, The Netherlands, April 2002.

[19] Sato, Y., Chiba, S. and Michiaki, T. *A Selective, Just-in-Time Aspect Weaver.* In Proceedings of the second International Conference on Generative Programming and Component Engineering. Erfurt, Germany, September 2003.

[20] Serini, D. and De Moor, O. *Static analysis of aspects.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[21] Suvee, D., Vanderperren, W. and Jonckers, V. *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[22] Vanderperren, W. *A pattern based approach to separate tangled concerns in component based development.* In Proceedings of ACP4IS workshop at AOSD 2002. Enschede, The Netherlands, April 2002.

[23] Vanhaute, B., De Win, B. and De Decker B. *Building Frameworks in AspectJ.* Workshop on Advanced Separation of Concerns.

[24] Verheecke, B., Cibran, M. A. and Jonckers, V. *AOP for Dynamic Configuration and Management of Web services in Client-Applications*. In Proceedings of 2003 International Conference on Web Services. Erfurt, Germany, September 2003.

[25] Verspecht, D., Vanderperren, W., Suvee, D. and Jonckers, V. *JAsCo.NET: Unraveling Crosscutting Concerns in .NET Web Services.* In Proceedings of Second Nordic Conference on Web Services NCWS'03. Vaxjo, Sweden, Novermber 2003.

[26] Workshop on “feature interaction in composed systems” at ECOOP 2001. Program available at http://www.info.uni-karlsruhe.de/pulvermu~/workshops/ecoop2001.

[27] Wydaeghe, B. and Vanderperren, W. *Visual Component Composition Using Composition Patterns.* In Proceedings of Tools 2001. Santa Barbara, USA , July 2001.