Just-In-Time Aspects: Efficient Dynamic Weaving for Java*

Andrei Popovici, Gustavo Alonso, and Thomas Gross
Department of Computer Science
Swiss Federal Institut of Technology Zürich
CH-8092 Zürich, Switzerland
{popovici,alonso,trg}@inf.ethz.ch

ABSTRACT

Recent developments in service architectures suggest that run-time adaptations could be implemented with dynamic AOP. In this paper we discuss application requirements on run-time AOP support and present a system that addresses these requirements. We provide basic support for weaving using the Just-In-Time compiler, while the AOP system is treated as an exchangeable module on top of the basic support. This approach allows us to provide a low run-time overhead, AOP system flexibility, and secure weaving. We provide an extensive empirical evaluation and discuss the trade-offs resulting from using the JIT compiler and a modularized architecture.

1. INTRODUCTION

Aspect-Oriented Programming (AOP) [8] holds the promise of composing software out of orthogonal concern spaces [20]. More recently, there has been a growing interest in using dynamic aspect-oriented techniques [13, 11, 3, 16, 12] to express run-time adaptations of services.

In our research [18, 15] we have encountered a number of design problems that can be addressed by using dynamic AOP. A first example is hot fixes in web services. A hot-fix is an extension applied to a running application server to modify the behavior of a large number of running components. Hot fixes can be used for software patches, security breaches, dealing with unexpected changes in network traffic, server availability, or providing client-specific services [21]. A second example is adaptation of mobile devices. In this case, dynamic AOP offers flexibility and simplicity over existing solutions based on reflective middleware [5, 2].

A potential drawback of dynamic AOP is the performance overhead. In addition, dynamic AOP may be insecure since it can allow weaving of malicious advice code. Finally, existing solutions for dynamic AOP are monolithic systems. They do not cleanly separate the join-point model (which represents the actual *support for AOP*) from the *AOP system* (we use the term AOP system in the sense of [6]). Some applications, however, require to use their own AOP systems on top of an existing join-point model. Solving these limitations would help dynamic AOP to become more widespread. In this paper we address these questions by:

- formulating a number of requirements to dynamic AOP systems based on relevant use-cases,
- addressing these requirements with a modular system architecture that separates the support for dynamic AOP (provided by the just-in-time compiler) from the AOP system, and
- presenting a prototype that identifies the cost of such system (in terms of execution overhead and implementation complexity).

The paper is structured as follows: In Section 2 we describe several requirements on dynamic AOP. In Section 3 we define the basic architecture of the system. The implementation of the Java Virtual Machine (JVM) support for aspect-orientation is explained in Section 4. Section 5 illustrates the design of the complete system including the instruments needed for weaving aspects. The experiments involving dynamic AOP in a Jini network are described in Section 6. We conclude the paper in section 7.

2. MOTIVATION

2.1 Run-time support for mobile computing

Most mobile devices have the constraint of limited storage space. This implies that a mobile device cannot be provided from the start with all software components needed in various locations. To overcome this problem, the mobile device must dynamically acquire the functionality it needs to operate in a certain location and discard this functionality when it changes location. This functionality has often a cross-cutting character. Hence the ability of acquiring and discarding cross-cutting functionality upon joining or leaving a network would be beneficial for many applications.

Illustrative examples are large conferences, trade shows, business meetings, or exhibition halls where participants are provided with computing devices such as lap-tops, desktops or PDAs. Customers may want to buy products from retailers after consulting their electronic catalogs. Retailers and customers may want to keep a log of their electronic transactions for later reference and settling of disputes, etc. Our main objective with the implementation of such scenarios is that participants should not be required to carry

*Effort sponsored in part by the Swiss National Science Foundation NCCR MICS (Mobile Information and Communication Systems).

with them all the software needed for security, privacy, data persistence, logging, arbitration of electronic transactions, etc. This functionality should be a property of the environment and should be available upon demand. Using current technology, this is difficult to achieve. The problem is that modern information systems (e.g., Enterprise Java Beans [9]) combine middleware and application logic at deployment time. As a consequence, all participants would have to stop their devices in order to install the location-specific software components before they could resume their work. What is needed is a service infrastructure equivalent to network-wide containers where important functionality for transactions, persistence and security can be dynamically woven through all mobile services joining the network and unwoven from all services leaving the network. To achieve this, each mobile device must carry a dynamic AOP system. Using this system, it can receive aspects that implement the middleware functionality of the current network. A preliminary evaluation of a network container prototype [15] revealed two important requirements:

Requirement 1: Efficiency under normal operations. Under normal operations (no woven aspects), the dynamic AOP system should not lead to significant performance degradation.

Requirement 2: Secure and atomic weaving. First, run-time weaving must be secure: one should be able to control what local resources can or cannot be used by advice code originating from foreign hosts. Second, the weaving operation should appear to the application as one atomic step. Intuitively, imagine an aspect that adds advice around 100 methods of a running base application. If the weaving operation is not atomic, the advice is added one join-point at a time. This may lead to a situation where half of the methods execute the additional functionality, while the other half still uses the old base code.

2.2 Hot fixes for application servers

HEDC, HESSI Experimental Data Center, [18] is a multi-terabyte repository built for the recently launched HESSI satellite. HESSI observes the sun and builds catalogs with events of interest such as sun flares. In HEDC, scientific users are confronted with large catalogs they need to browse and update. At a certain point in time, an older servlet-based web service for browsing the database was re-activated. Since the main system had evolved in the meantime, the web service resulted in performance degradation for all users (including those who were not browsing the catalog via web). The analysis identified the problem: for each http request, a *session object* was created that incurred a significant workload on the database server.

Fixing the problem (adding pooling for session objects) revealed how useful aspect-oriented *hot fixes* can be. First, the problem has a clear cross-cutting concern: it requires replacing code (e.g., `new Session()`) which is scattered through multiple servlet classes with code that reuses session from previous invocations. Second, in a service environment, the load on the database can be decreased considerably when the first corrections are applied without taking the service off-line. Third, the system was based on a proprietary library (the source code was not available). Therefore, a way to apply the fixing aspect without relying on the source code would have given the necessary time to obtain the sources and re-factor the system properly.

The HEDC case is not exceptional, and many other applications face similar problems [21]. Assuming that each application server would have an AOP system capable of dynamic weaving, this and related problems could be solved efficiently. To achieve this, a number of issues must be addressed by such a system:

Requirement 3: Efficient advice execution. Executing advice functionality needs to be done efficiently, as these systems typically exploit the available resources (CPU) to the maximum.

Requirement 4: Flexibility. The application should allow exchanging the AOP system. This would allow developers to use the most appropriate AOP system for each case (e.g., an AOP system that addresses the particular types of cross-cutting concerns required by the fix or the AOP system best known by the programmer).

3. BASIC ARCHITECTURE AND GOALS

3.1 Addressing requirement 1

There are two basic levels where to locate the dynamic weaving support in a Java-based environment:

- at the class-loader level, by transforming the byte-code before it is loaded in a JVM,
- at the just-in-time (JIT) compiler level, by inserting the advice directly into the native code generated by the JIT compiler.

Existing dynamic AOP systems like JAC [13] and Handi-Wrap [3] enhance the original code with minimal hooks that enable dynamic weaving. Run-time weaving of an aspect A actually corresponds to the activation of those hooks matched by A. Because hooks are woven at *all* potential join-points (e.g., method calls, method executions, field sets, field gets), the impact of the hook code must be kept minimal, even when no aspects are specified.

In this respect, we expect hooks woven at byte-code level to cost more than those woven at JIT level (the reason being that native code can be much more optimized). For example, the results published on Handi-Wrap show an overhead of around 10% for method boundary join-points. Our initial performance assessment of the JIT solution indicated that the same overhead of 10% can be expected for enabling join-points at both field sets, field gets and method boundaries. Hence, using a JIT-based weaver seems to be a promising way of maintaining a low overhead under normal operation (requirement 1).

3.2 Addressing requirement 2

A JIT-based weaver can weave minimal hooks (like JAC and Handi-Wrap) or weave the actual advice code (like AspectJ [22] does at compile-time). With respect to requirement 2, each option has advantages and disadvantages.

Minimal hook weavers. Hook-based weavers can avoid security problems (weaving of malicious advice code) in a straightforward way. When hooks are woven through the base code, the actual advice code can be kept separate from the base application. This separation fits well with existing

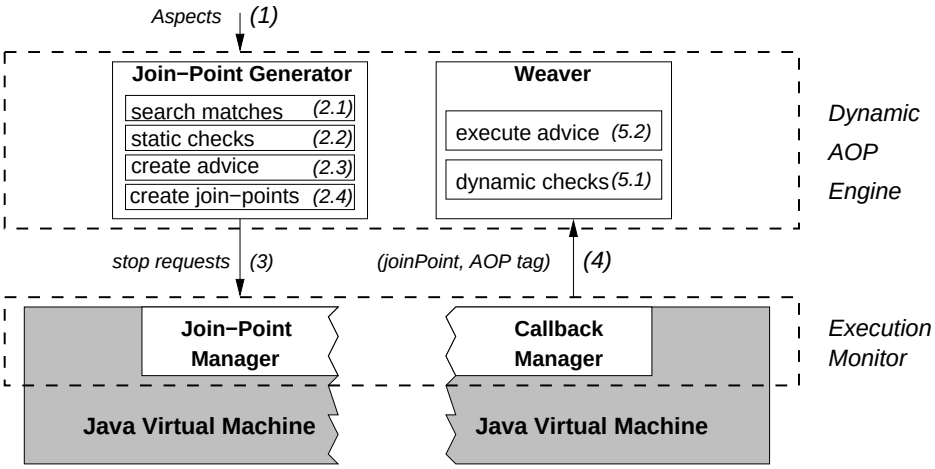

Figure 1: Two-layer architecture of the AOP support in a JVM.

security models that control the access to system resources based on the origin of entire classes or archives (e.g., the Java Security Model [19]).

Atomic weaving can be easily implemented in a hook-based weaver. This is possible because of the additional indirection layer represented by hooks: every time when a join-point is reached, a hook method is called, not the actual advice. Weaving atomicity can be achieved by blocking advice execution in the hook method until the weaving operation completes.

Advice weavers. Alternatively, the JIT compiler could weave the actual advice code. This approach may result in more concise advice. By contrast, the advice added by hook weaver may have to access the local environment of a join-point in a reflective way [16]. An advice weaver is also faster than hook-based weavers because it avoids indirections. However it increases the complexity of JIT support to address requirement 2: weaving advice code makes re-JITing necessary once an aspect is removed or exchanged. It moves the responsibility of dealing with atomic advice activation to the JIT-level, thereby increasing the complexity of the JIT compiler. Finally, the advice code would not be separated from the original code. This would hamper direct usage of the Java Security Model, which would have to be integrated within the JIT-compiler.

From these ideas, we opted for a minimal hook weaver which inserts hooks at all potential join-points.

3.3 Addressing requirement 3

For the same reasons explained above we expect the overhead incurred by the JIT – when actually calling an empty hook – to compare favorably with the overhead incurred by other approaches. With this in mind, a JIT-based system addresses requirement 3. We show such a comparison in section 5.2.

3.4 Addressing requirement 4

A JIT-based weaver trades portability for performance. The native advice code is smaller and more efficient, but the system is tightly coupled with a particular JVM and JIT version. To solve this problem, we propose a clean separation between the aspect-support embedded in the JVM, (the ***execution monitor***) and the application specific AOP system (the ***dynamic AOP engine***). The execution monitor exports a Join-Point Model API to the engine. By doing this, it provides the basis for exchangeable AOP engines, thereby addressing requirement 4.

3.5 Architecture

Figure 1 gives an overview of this type of architecture. In the upper layer, the AOP engine accepts aspects (1) and transforms them into basic entities like join-point requests (2.1 - 2.4). A join-point request is a *description* of the code location where the execution must be interrupted in order to executed advice. It activates the join-point requests by invoking methods of the execution monitor (3). The execution monitor is integrated with the JVM. For example, when the execution reaches one of the activated mjoin-points, the execution monitor notifies the AOP engine (4) which then executes an advice (5).

The main focus of the execution monitor is to address the requirements 1 and 3.

The design of an AOP engine is much less constrained than that of an execution monitor. Its responsibility is to define the mechanisms for atomic and secure weaving (requirement 2). Behind the *Weaver* interface, the AOP engine can hide its platform-specific aspect features, thus providing the flexibility formulated by requirement 4.

Our goals when developing this system were to:

- define a clean interface of the execution monitor, suitable for implementing dynamic weavers on top of it, while reducing the requirements to the execution monitor. This goal motivates the decision to weave minimal hooks and not advice code (because weaving minimal hooks reduces the complexity of the JIT support). The idea would be for JVM vendors to provide this interface at a low (implementation) cost.
- propose a join-point API similar to the AspectJ model, to encourage exchanging the AOP engine built on top of the execution monitor,
- provide an initial evaluation of the performance and implementation complexity of such a system.

```
interface JoinPointManager {
  // init
  void aopScope(String[] prefixes, boolean openWorld)

  // register join-points
  void watchMethodEntry ( Method m, Object aopTag )
  void watchMethodExit ( Method m, Object aopTag )
  void watchFieldAccess ( Field f, Object aopTag )
  void watchFieldModification ( Field f, Object aopTag )

  // unregister join-points
  ...
}
```

(a)

```
interface Weaver {
  // requested join-points
  void onMethodEntry ( JoinPoint jp )
  void onMethodExit ( JoinPoint jp )
  void onFieldAccess ( JoinPoint jp )
  void onFieldModification ( JoinPoint jp )

  // class loads
  void onClassLoad(Class cls)
}
```

(b)

Figure 2: (a) The JoinPointManager API and (b) the callback methods called by the execution monitor.

3.6 The execution monitor

Figure 2 contains the core interface of the execution monitor. The execution monitor is integrated with the JVM. It contains the functionality for activating join-points (used when a new aspect is added to the system) and the callback functionality for notifying the AOP engine that a join-point has been reached (used at run-time).

The execution monitor intercepts the execution in all user and system classes. Because of security or performance reasons, it is sometimes desirable to restrict the scope of dynamic AOP to a subset of all possible classes. The method aopScope in Figure 2.a can be called at start-up to specify the interception scope. It receives a list of strings, representing class-name prefixes, and a boolean parameter openWorld. If openWorld is true, then all classes, except those prefixed by one of the prefixes are subject to interception. A false value implies that interception is performed in a closed world, consisting of the classes that match one of the prefixes.

When a new aspect is added to the system, the AOP engine *activates* join-points using the watch* methods of the JoinPointManager. The first parameter is usually a class member (e.g., field or method) that uniquely identifies the join-point. The second parameter is the aopTag, a client-side data object that will be passed to the weaver callback. The aopTag may contain raw byte-code or data. The semantics of the AOP tag are irrelevant for the execution monitor. When an activated join-point is reached, the callback manager notifies the AOP engine (each AOP engine implements the Weaver interface, depicted in Figure 2.b). Notification is suppressed if the AOP engine is not set up (this is the case if applications employ the JVM in the traditional mode).

The join-point argument jp contains methods for the inspection of local variable values, thread states, return values, etc. Additionally, the execution monitor guarantees that the aopTag specified at join-point activation is also part of jp's state.

The execution monitor interface has a reflective character (e.g., local variables are first-class Java entities). However, the aim of the execution monitor is not that of a full-fledged Meta-Object Protocol (MOP), which usually exposes much more program and execution constructs than are needed for AOP. The reflective character of join points can be hidden by the AOP engine.

3.7 The AOP engine

When a new aspect is woven, the *join-point generator* decomposes an aspect into join-point requests and activates join-points over the JoinPointManager API. When an active join-point is reached, the weaver executes a corresponding advice. The weaver and the join-point generator define the AOP engine together. By replacing these two components, one can define new AOP systems on the same JVM.

We explain the general architecture of the AOP engine by illustrating the actions for weaving an aspect using an example AOP engine. Then we show the actions taken to execute the corresponding advice. Our example aspect definition encrypts all bytes-array parameters passed to sendBytes-methods. We use pseudo-code to abstract from a particular AOP dialect:

> **before** methods-with-signature *'void *.sendBytes(byte[] x)'*
> **do** encrypt(x)

To weave this aspect, the join-point generator performs several sub-tasks, like in Figure 1. First, it inspects all the classes currently loaded by the JVM and gathers all methods $m_1..m_n$ that match the signature 'void *.sendBytes(byte[] x)' (Figure 1, step 2.1). It then performs static checks, e.g., ensures that the method encrypt exists, and that the formal parameters of encrypt are assignable from a byte-array (step 2.2). Thirdly, it defines the client data to be passed back by to the weaver when the join-point is actually reached. In this example, the client data is the method doAdvice, which will contain an invocation to encrypt. (step 2.3). For all generated join-point requests (step 2.4), it activates method entry join-points (step 3). As a value for aopTag it specifies an array containing the code of doAdvice or, alternatively, a reference to the memory location where it resides. The semantic of the aopTag parameter is not important for the execution monitor, but it will be very useful when calling back the weaver.

Because of dynamic class loading, aspect weaving is more complex in practice. When a class is dynamically loaded, the join-point generator applies the aspects already loaded in the system to the newly loaded class. It then activates the join-points belonging to the loaded class.

The example aspect denotes join-points that are all captured by static points in the original program code (method entries). In practice, aspects are defined both by static and by dynamic criteria. It is the responsibility of the AOP engine to filter out the JoinPoint objects received from the execution monitor if they do not correspond to the original aspect definition.

To exemplify these principles, we describe in Figure 3 a simplified implementation of the weaver. Upon entry in $m_1..m_n$, onMethodEntry is called by the execution monitor. The weaver guarantees that all run-time conditions defining the reached join-point are met (line 4). If all the dynamic

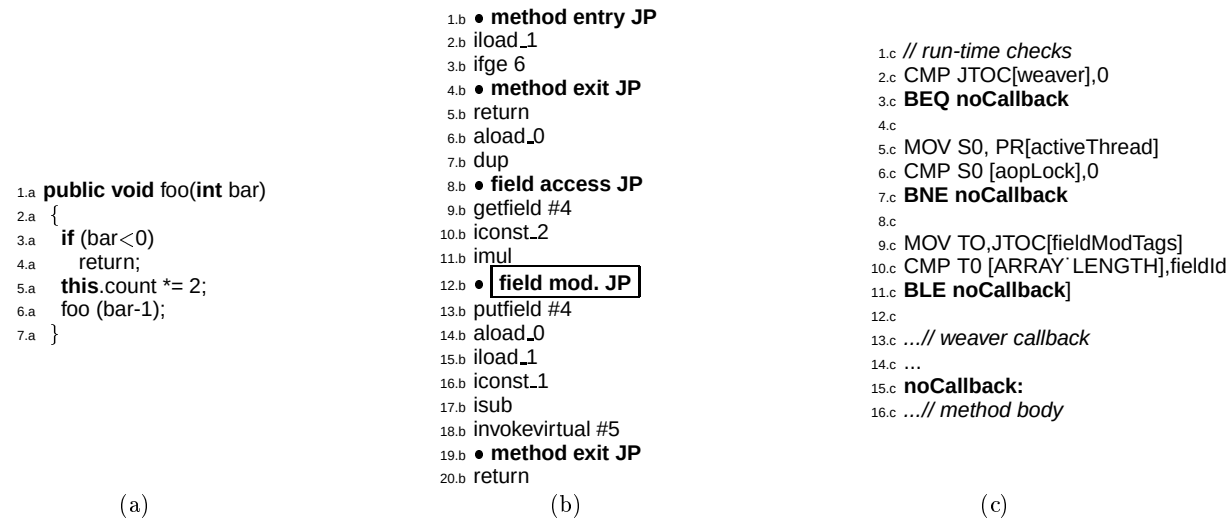

Figure 4: (a) example Java code (b) Translated byte-code (c) Join-Point stub for the field modification.

```
1 void onMethodEntry(JoinPoint jp)
2 {
3   // 5.1: dynamic filtering
4   dynamicChecks();
5
6   // 5.2: advice execution
7   byte[] codeToExecute = jp.aopTag;
8   executeInternal(codeToExecute);
9 }
```

Figure 3: Simple implementation of a Weaver method.

checks are passed successfully, the advice can be executed. On line 7, the weaver extracts the code array stored in the aopTag. This code corresponds to the inspectStackAndCallEncrypt method. Finally, the weaver interprets or executes the method.

The rest of the methods of the Weaver interface can be implemented analogously. Because of space reasons, we do not describe here the actions needed for unweaving aspects.

4. THE JIKES EXECUTION MONITOR

Our implementation of the execution monitor is based on the IBM Jikes Research Virtual Machine [1]. Jikes is written in Java, and employs a compile-only strategy: all methods, including those belonging to the JVM itself, are translated to native code before execution. To implement the AOP support, the baseline (non-optimizing) compiler of Jikes, version 2.0.2, was modified. This section first presents the enhancement of the JIT compiler, then the additions to the JVM, and finally the performance results of the AOP-aware JVM.

4.1 JIT compiler enhancements

To make join-points interceptable, the JIT compiler weaves minimal hooks (henceforth called *join-point stub* instructions) at native code locations that correspond to join-points. This operation is done conditionally, depending on the initialization parameters passed to aopScope.

Figure 4 illustrates how a join-point stub is generated. A small Java method is described in Figure 4.a; its byte-code representation is contained in Figure 4.b. At all places marked with a bullet, the just-in time compiler generates the join-point stub code. Figure 4.c is the join-point stub for the field modification. Before giving the control to the execution monitor, the join-point stub checks that the AOP engine exists (lines 1.c-3.c), that the weaver can be called safely (lines 5.c-7.c) and that the join-point is activated (lines 9.c-11.c). In many cases, it jumps directly to the beginning of the method body (label noCallback on line 15.c).

Lines 5.c to 7.c check whether the advice action may trigger AOP recursion. Recursion occurs if, during the execution of an advice action a_1, a join-point triggers a new advice action, b_1 which hits a join-point that executes a_1 once again. AOP recursion is harder to understand and use than classical recursion. One join-point may trigger several advice actions $a_1..a_n$; the same advice may be triggered by different join-points. Moreover, in dynamic AOP, the effects of aspect-oriented recursion depend on additional run-time parameters, e.g., the point in time when an aspect was added to system, or the set of currently woven aspects. Our approach is to lock the join-points on a per-thread basis during advice execution. This effectively disables AOP recursion. A less restrictive, but more costly approach, would be to detect recursion cycles for each join-point dispatch.

For obvious reasons, the run-time checks must be very efficient. For the large majority of join-points, an overhead of 2 to 6 machine instructions is expected. If the join-point "belongs" to an aspect, the callback method of the weaver is called. The callback is expensive: all data characterizing the join-point (e.g., local environment, the aopTag specified at join-point registration) is packed into a JoinPoint object which is then passed as a parameter to the weaver.

4.2 JVM Enhancements

The generation of join-point stubs changes a number of properties on which the execution of the VM relies. For consistency and efficiency reasons, the integration of the execution monitor with several components of the JVM is needed.

One example is the division of a method's body into basic blocks. A basic block is a code sequence in which the stack layout remains unchanged. Basic blocks are used by the garbage collector to inspect the stack and collect object references. When adding join-point stubs, a method call may

Benchmark	Relative overhead
Java Grande benchmark suite	
LUFact:Kernel	103.15 %
Crypt:Kernel	103.24 %
SOR:Kernel	98.74 %
SparseMatmult:Kernel	100.23 %
Average	**101.44** %
SPECjvm 98 benchmark suite	
check	103.04 %
jess	110.19 %
db	105.17 %
jack	107.84 %
javac	113.51 %
Average	**107.95** %

Table 1: Relative overhead with AOP support for method boundaries, field sets, and field gets.

occur at a location where no basic block boundary was detected during byte-code analysis. As consequence, JIT-level AOP implies the additional cost of adapting the byte-code analysis component.

Another problem that we face is making the stack layout visible to the join-point object passed as a parameter to Weaver. This is needed because the JoinPoint interface allows gathering information of the local environment (e.g., local variables).

In related approaches [10], it was observed that the generation of reflective information (here, the join-point object) has an important impact on performance and on the frequency of garbage collection. To minimize this impact, the internal thread data structure can be enhanced to contain a pool of join-point objects. When calling back the AOP engine, the generation of new objects is thus avoided. In general, all information related to join-points must be as close as possible to the internal representation in the JVM.

4.3 JVM performance

The integration of the execution monitor implies minor changes to the JVM classes (600 lines of code) plus an additional module of approximately 1000 lines.

We first compare the original JVM with the JVM containing the execution monitor. In this experiment, the execution monitor is not activated. The results measure the performance loss incurred by the mere existence of the AOP support and join-point stubs. All experiments were performed on Linux, running on a Pentium III 500MHz double processor machine with 512 MBytes RAM.

Table 1 summarizes the relative overhead of the AOP enhanced JVM for the Java Grande [7] and SPECjvm [17] benchmarks. The AOP support leads to a slowdown of averagely 1.5% in Java Grande tests. An average slowdown of 8% is observed in the SPECjvm tests.

To measure the code size variation incurred by various join-point stubs, we compare the size of the native code generated by the baseline compiler with and without AOP support. The code belongs to a number of 2593 methods in the core libraries of the JVM. Table 2 shows the relative code-size increases, as a function of the join-point type.

The unmodified baseline compiler translates a method in 310µs. The aop-baseline compiler takes 380µs for compiling a method and produces code which is twice as large

Type of join-points	New code size
Method entry:	126.3 %
Method exit:	124.9 %
Field modification:	111.8 %
Field access:	61.1 %
Total:	**233.9** %

Table 2: Relative increase of the code size due to weaving of join-point stubs.

(Table 2). The slowdown of the compilation depends heavily on the aspect scope specified at initialization. For each translation of a method m declared in class C, the compiler verifies that C is in the aspect scope. For an aspect scope which is defined by a large list of package prefixes, the compilation time grows accordingly.

To measure the efficiency of the execution monitor, we performed a number of *micro-measurements*. For a micro-measurement, we calculate the time needed to execute a simple operation (e.g., an empty method call, a field set, a field get). The example below illustrates how we measure the cost of an empty method call:

```
1 for(i=0;i<100000;i++); // t1: loop time
2 for(i=0;i<100000;i++) tstMethod(); // t2: loop + invokevirtual
3 // time needed for an meth. invocation: (t2 - t1)/100000
```

When doing the same measurements with an activated join-point (e.g., before the execution of tstMethod) the time increases accordingly. This increase indicates how much time it is spent in the execution monitor and is a good indicator of the AOP system efficiency. We use the common technique of a trivial weaver implementation with "do-nothing" operations for each type of join-point. We repeat the same measurement until the standard deviation is less than 1%[14].

The micro-measurement results are summarized in Table 3. Each row contains the average time needed to execute a byte-code instruction, under various configurations of the AOP support.

On the first column we list the four basic operations we have evaluated. The second column represents the time needed to execute an instruction when the execution monitor has no active join-points. The third column contains the cost of executing an operation for which a join-point was registered and then locked. Recall that join-points are locked to avoid AOP recursion, hence the third column shows the cost of reaching a join-point *during* an advice action.

The fourth column contains the execution time of a byte-code with an activated join-point. An activated, unlocked join-point always results in a call to the weaver component. In our case, the weaver executes a do-nothing operation. The cost is significant: 500ns/instruction, roughly the time needed to execute an invokeinterface instruction. For comparison, a do-nothing AspectJ (version 1.0.3) advice, compiled into the test increases the execution time of an instruction by roughly 100ns (column 5).

Finally, an important parameter is the cost of accessing objects in the local environment of a join-point using the JoinPoint interface. The largest cost (1110 ns) is for retrieving local integers. For retrieving the "this" object 380ns are needed, while one of the actual parameters of type Object can be accessed in 460 ns. The difference between retrieving

Instruction type	No call to weaver because join-point is not activated	No call to weaver because join-point is activated but locked	Call to empty weaver method because of an active, unlocked join-point	AspectJ call to an empty advice
getfield	12.9 ns	17.7 ns	541.3 ns	108.3 ns
putfield	12.3 ns	19.5 ns	548.0 ns	119.7 ns
invokevirtual	39.0 ns	47.9 ns	513.9 ns	201.3 ns
invokeinterface	662.6 ns	670.1 ns	1121 ns	850.1 ns

Table 3: Join point costs on the enhanced JVM (columns 2-5) and on the unchanged JVM with AspectJ and a do-nothing advice.

objects and integers is simple to explain: the JoinPoint interface does not return primitive values (e.g., int) but wrapper objects (e.g., java.lang.Integer). The rest of the time is spent to locate the local variable in the current stack frame. The cost of accessing parameters has been reported to be significant in related approaches [3] (also based on minimal hooks). In our architecture, the cost of retrieving variables from the stack depends on the implementation of the AOP engine. One AOP system may be eager, and always retrieve all visible variables from the stack when reaching a join-point. Another one may be lazy, and retrieve the variables only when the advice code action needs them. Which option is better still needs to be explored.

4.4 Evaluation of the execution monitor

The execution monitor prototype gives us some important information about how this architecture fits the initial requirements and conveys a first indication of the performance that can be achieved by a JIT-based weaver.

If a 5% to 10% slowdown is acceptable for applications that need dynamic AOP, then the requirement 1 is well addressed with this model. For example, in Java Grande tests, the enhanced JVM leads to a small slowdown, of only 1.5%, while the SPECjvm tests show an average of overhead 8%. The difference is probably due to the fact that Java Grande applications are computationally intensive and contain fewer join-points than the SPECjvm tests. The performance overhead incurred for join-points reached during the control flow of an advice cost less than 9ns.

The largest cost is incurred by active join-points: the notification of the AOP engine is roughly equivalent to the cost of an invokeinterface call (0.5 µs). A join-pointed instruction is 1.3 to 5 times slower than a static implementation based on AspectJ. A higher run-time cost than static AOP lies in the nature of dynamic AOP. As suggested by our initial performance assessment, the execution monitor compares favorably to other approaches. For example, our first prototype [16], based on the debugger interface of the JVM, needs 101.1µs for an upcall before an invokevirtual and 98.3 µs for an upcall before an invokeinterface instruction. This corresponds to a relative overhead between 16 and 106 times larger than the overhead incurred by the JIT execution monitor for a similar operation.

The implementation cost of an execution monitor is small (1600 loc). The small implementation cost is also due to the decision to implement the system using the baseline JIT compiler. An implementation based on the optimizing compiler is more difficult, because the existence of join-points may prevent certain compiler optimizations. This may lead to a larger impact on the execution time of applications. In both cases, an efficient execution monitor requires JVM-specific knowledge, since it has to be integrated with core modules of the JVM (e.g., the garbage collector, internal thread structures).

5. THE AOP ENGINE WITH PROSE

5.1 PROSE

To check the feasibility of our model, we wanted to evaluate the cost of a complete AOP system. For this purpose, we adapted PROSE [16] to use the execution monitor and join-point API. PROSE consists of a set of libraries. Aspects in PROSE are first-class Java entities, and all related constructs are expressed using the base language, Java.

The use of PROSE is best shown by means of an example. A PROSE aspect for implementing access control in all methods with names matching "m*" and belonging to classes named ServiceB is depicted in Figure 5. All PROSE aspects extend the Aspect base class (line 1). An aspect object contains one or several crosscut objects. A crosscut object[1] defines an advice and describes the join-points where the advice should be executed. In Figure 5, there is just one crosscut, corresponding to the `accCtrl` instance field (line 3); the advice action is defined on the lines 6-9. The number and types of join-points defined by `accCtrl` depend on the signature of the advice method and on a *specializer* object attached to the crosscut (lines 12-14). The signature (line 6) restricts the execution of the advice to methods declared in classes of type ServiceB. The specializer further restricts the set of join-points to entries in methods whose name matches the regular expression "m.*". Specializers are composable by means of NOT,AND, respectively OR, methods. They are used in a way that is similar to that of *pointcut designators* [22] in AspectJ.

We adapted the initial implementation of PROSE [16] to use the execution monitor interface. Since aspects are first-class Java entities, no parsing of aspects is implemented by PROSE's join-point generator. Nevertheless, the other subtasks of the join-point generator component are present: static checks verify the compatibility of the advice formal parameters with those of various intercepted methods; and join-points are generated and then activated in the JoinPointManager.

PROSE also defines its own Weaver implementation. The PROSE Weaver performs dynamic filtering of join-points depending on the specializers associated with a crosscut object. When calling an advice method like ANYMETHOD (line 6) PROSE converts the actual parameters of the intercepted method to types required by the advice method interface (e.g., REST). Lastly, it calls the advice method.

[1]In this context, a crosscut is a programmatic construct roughly equivalent to the advice construct in AspectJ.

Invoke type	Parameters	Jikes/AspectJ	Jikes/PROSE
invokeinterface	(Object,Object)	969 ns	2691 ns
invokeinterface	()	528 ns	2180 ns
invokevirtual	(Object,Object)	203 ns	1746 ns
invokevirtual	()	201 ns	1783 ns
sync invokevirtual	(Object,Object)	483 ns	2020 ns
sync invokevirtual	()	471 ns	1981 ns

Table 4: Micro-benchmark results with PROSE and AspectJ.

```
1  class SecurityAspect extends Aspect
2  {
3    Crosscut accCtrl = new MethodCut()
4    {
5       // advice method:   ServiceB.*(..) && ...
6       public void ANYMETHOD(ServiceB thisO, REST anyp)
7            {
8                // check access SeviceB.*
9            }
10      // ...&& before m*(..) && instanceof(Remote)
11      { setSpecializer(
12           (MethodS.BEFORE)          .AND
13           (MethodS.named("m.*"))    .AND
14           (TargetS.inSubclass(Remote.class)) );
15      }
16   };
17 }
```

Figure 5: A PROSE aspect for weaving location-specific access control at the start of methods defined in ServiceB.

One important issue is the support for atomic weaving (requirement 2). Atomic weaving corresponds to the activation of join-points matched by an aspect A in one single step. The PROSE engine provides this support as follows. It activates join-points one by one (non-atomically) but sets a flag in the A's advice method that makes its execution a do-nothing operation. As more join-points are activated, A's advice *is* actually invoked, but it has no visible effect at run-time. Once all the join-points corresponding to A's advice have been properly activated in the execution monitor, the engine unsets the flag (field sets are atomic in Java). From this point on, reaching a join-point matched by A is followed by the execution of the actual advice.

5.2 Evaluation of the AOP Engine

We repeated a number of micro-measurements done for the execution monitor, this time with the complete AOP system (the PROSE AOP engine running on top of the Jikes execution monitor). Here, too, we measure the cost of do-nothing advices around method invocations. Table 4 summarizes the results. Each line contains the total time needed to execute a method call plus an additional do-nothing advice on method entry. The advices were woven statically using AspectJ (column 3) and dynamically using PROSE/Jikes (column 4).

The cost of executing a do-nothing PROSE advice is now 2.5 to 8.5 times higher than that of executing a static advice. It is fair to say that PROSE's AOP engine induces a large overhead compared to AspectJ. The complete weaver (execution monitor plus AOP engine) is significantly faster than the previous version of PROSE, based on the debugger interface of the JVM. Thus, the relative overhead at method boundaries is between 16 and 151 times smaller than in the previous version of PROSE.

We performed an additional set of micro-measurements to identify the difference between our JIT-based AOP system and a load-time based AOP system. Performance measurements published on [13] and Handi-Wrap [3] indicate that there are important performance variations among load-time weavers (e.g., Handi-Wrap is significantly faster than JAC). With JAC being publicly available, we chose to micro-measure the *relative overhead* induced by the two platforms (ours and JAC). By relative overhead we mean the fraction

$$t_{simple\_operation+advice\_action}/t_{simple\_operation}$$

We first compared the Jikes execution monitor and JAC when executing a "do-nothing" advice action before a known method. The relative overhead of the execution monitor for virtual (interface) calls was 780 (10873) smaller than JACs. This is however an unfair comparison, since we are comparing the support for the join-point model with a whole AOP System.

A more relevant comparison is between "do-nothing" advices in PROSE system and int the JAC system. In this case, the relative overhead of our system for a virtual (interface) method call is 233 (4180) times smaller than the one incurred by JAC. For these measurements we use the micro-measurement methodology explained earlier [14].

This difference is an important information, even with the significant performance differences among various bytecode weavers. It must be noted that JAC uses a different approach, with a more powerful join-point model which allows per-object advices. This join-point model may lead to improved performance in more specialized cases. However, the exact cause of the costs involved by the JAC join-point model and a careful analysis of the difference between the two systems is beyond the scope of this paper and will be addressed as part of future work.

Note that over 75% of the overhead time is spent in the PROSE AOP engine, the rest being spent in the execution monitor. This means that an important part of the current cost can be eliminated by a more efficient AOP engine. For example, employing an advice calling schema similar to [4] or [3] (which pre-compiles the advice code, as opposed to PROSE) would lead to performance improvements.

Providing atomic weaving proved to be a fairly simple task in PROSE. We believe that the same technique can be applied to other AOP engines as well.

6. DEPLOYING THE AOP SYSTEM

We have evaluated the prototype in terms of complexity and performance, but not shown how the system can be used in practice. This is best illustrated by the deployment of the AOP system in a spontaneous container [15].

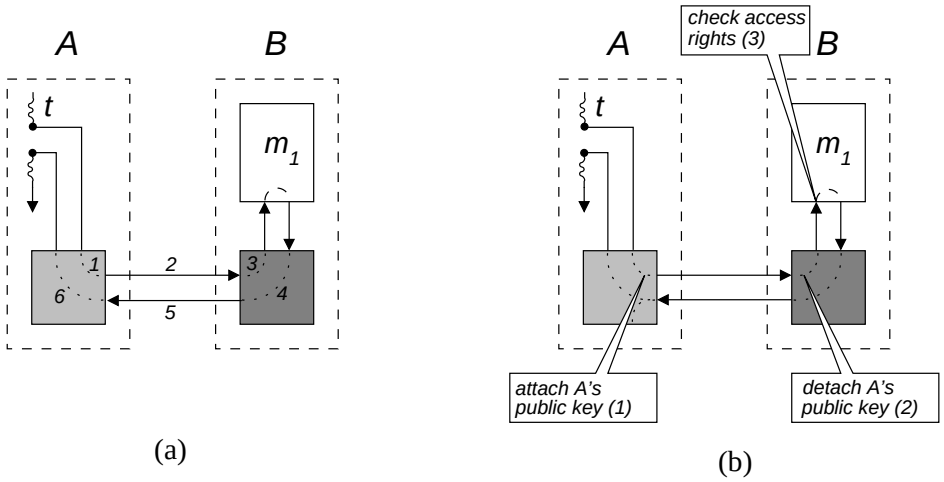

Figure 6: (a) Control flow during a remote call from A to B and (b) adaptation of node B with middleware specific features and (c) execution points enhanced by the security aspects.

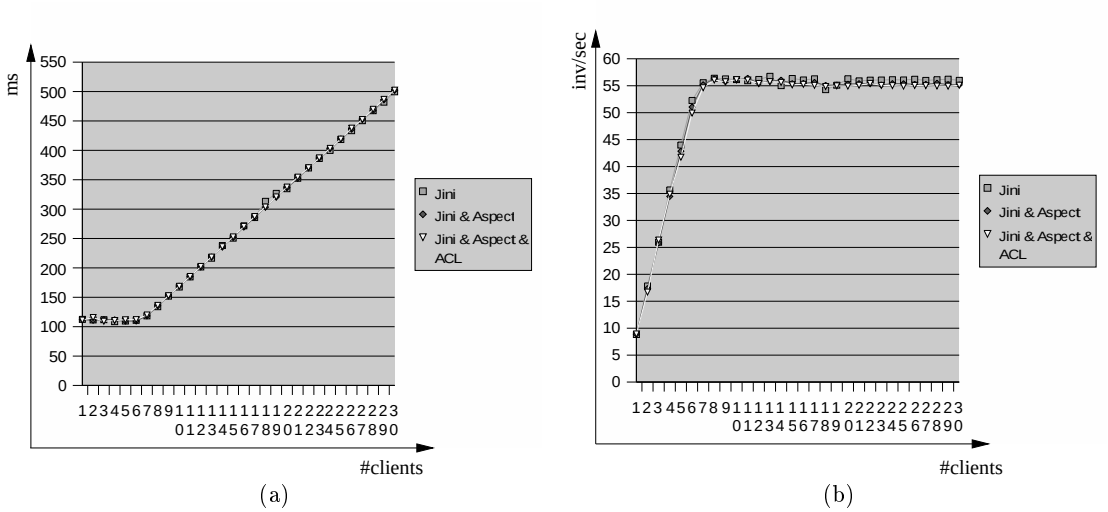

Figure 7: (a) Throughput in inv/s and (b) response time (ms) for remote invocations.

6.1 Basics of spontaneous containers

We consider the application scenario mentioned in the motivation section in which a spontaneous container provides dynamic middleware services to all devices on a fair-trade ground. A spontaneous container imposes two basic requirements to individual computing devices (nodes): First, each node exports a number of services, which can be remotely invoked from other nodes. Second, each node contains an activated dynamic AOP engine. Nodes receive aspects from a base station (the spontaneous container) and weave them at run-time using their AOP engine. In this experiment, the spontaneous container distributes aspects thus imposing a network-specific security policy to all nodes of a local network. We assume the aspects sent by the spontaneous container model access control in remote method invocations between any two nodes.

Figure 6.a illustrates the control flow of a remote service invocation between nodes A and B. On As node, the thread t calls an RMI stub (light gray). An RMI skeleton (dark gray) on B's site unmarshals the parameters and calls the actual implementation of the service method m_1. After m_1 has terminated the computation, similar steps are needed for transporting the return values form B to A. The return values are used for further computations in t.

Figure 6.c illustrates the same interaction, once the spontaneous container has sent aspects to both nodes A and B. After aspect weaving, the marshaling of parameters is intercepted on the callees site. At this point, the advice functionality adds A's public key to the list of actual parameters (1). On B's site, the unmarshaling is intercepted and the public key is extracted by an appropriate advice (2). Before the execution starts in m_1, A's access rights are checked (3). If the caller does not have the necessary rights, the execution is abruptly terminated.

6.2 Application performance

The AOP engine is a sub-component of a larger spontaneous container prototype [15]. The experiments describe

the behavior of a system in which a service B is remotely called by a variable number of clients $A_1..A_{30}$ (the clients work concurrently). For all experiments, we have grouped the client applications on one host of a local area network; the server runs on a different machine. Each client A_i calls the server 300 times. A remote call accepts a query image (40x40 pixels) as a parameter and returns the most similar image from the local database of images. The initial image database contains 60 images.

In all experiments, we measured two variables. The response time observed by each client is a good indicator of B's quality of service. The throughput (average number of invocations per second) observed by B indicates the scalability of the system. In total, we performed three experiments.

The first measured response time and throughput in a typical (non-adapted) service community. The second measured response time and throughput after the spontaneous container has woven a "do-nothing" aspect in each node. The do-nothing aspect intercepts all necessary calls, both in the RMI stubs and before the execution of m_1. This test characterizes the impact of the adaptation mechanism. The third experiment showed how the system behaves when the woven aspects implements key transfer functionality plus an access control check for each remote method invocation. Figure 7 summarizes the results of the experiments.

The performance of the dynamic AOP support translates to a barely observable difference in response time or throughput, hereby meeting the application requirements.

7. CONCLUSION

In this paper we have presented a modular and flexible architecture for dynamic AOP. The support for dynamic AOP is provided by the just-in-time compiler and exports a simple join-point API to the actual AOP system (the AOP engine); the AOP engine can be treated as an exchangeable module. The Just-In-Time approach strikes a reasonable balance between implementation complexity and performance: when no aspects are woven into a JVM, a relatively small overhead of 8% of the execution time can be expected for weaving method boundaries, field sets and gets. When aspects are woven, the cost of advice invocations is equivalent to an invokeinterface call. The AOP support can be completely disabled if needed; this feature, plus the reduced implementation complexity makes this approach viable in any JVM implementation. If this support were integrated with all JVMs, several challenging applications such as hot-fixes and adaptations in mobile computing would largely benefit from it. We have shown the trade-offs implied by such an architecture and actually described a prototype that implements a form of run-time adaptation for mobile services. As an open source project, PROSE is open for contributions, and can obtained from http://prose.ehtz.ch.

8. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] D. Arregui, F. Pacull, and J. Willamowski. Rule-Based Transactional Object Migration over a Reflective Middleware. In *Middleware 2001: IFIP/ACM Intl. Conf. on Distributed Systems Platforms*, volume 2218 of *LNCS*, pages 179–196, 2001.

[3] J. Baker and W. Hsieh. Runtime Aspect Weaving Through Metaprogramming. In *1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands*, pages 86–95, Apr. 2002.

[4] P. Bothner. Kawa – Compiling Dynamic Languages to the Java VM. In *Proc. of the Usenix Technical Conference*, New Orleans, June 1998.

[5] M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. In *Middleware 2001: IFIP/ACM Intl. Conf. on Distributed Systems Platforms*, volume 2218 of *LNCS*, pages 160–178, 2001.

[6] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[7] J. G. Forum. The Java Grande Forum benchmark suite. Accessible from http://www.javagrande.org.

[8] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Akşit and S. Matsuoka, editors, *In Proc. of ECOOP'97 Jyväskylä, Finland*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[9] S. Microsystems. Enterprise Java Beans Specification, Version 2.0, Aug. 2001.

[10] A. Oliva and L. Buzato. The design and implementation of Guaraná. In *Proc. of the 5th USENIX Conf. on Object-Oriented Technologies and Systems*, pages 203–216. The USENIX Association, 1999.

[11] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag.

[12] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proc. of ECOOP'2002*, Malaga, Spain, 2002. Springer.

[13] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A Flexible Solution for Aspect-Oriented Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, pages 1–24, Kyoto, Japan, September 2001. Springer Verlag.

[14] A. Popovici. Measurement Results and Methodology for AOSD03. http://prose.ethz.ch/Wiki.jsp?page=Measurements.

[15] A. Popovici and G. Alonso. Ad-Hoc Transactions for Mobile Sevices. In *Proc. of the 3rd VLDB Intl. Workshop on Transactions and Electronic Services (TES '02)*, Hong Kong, China, Aug. 2002.

[16] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In *1st Intl. Conf. on Aspect-Oriented Software Development, Enschede, The Netherlands*, Apr. 2002.

[17] Spec - The Standard Performance Evaluation Corporation. SPECjvm. Web access http://www.spec.org/osg/jvm98/.

[18] E. Stolte and G. Alonso. Efficient Exploration of Large Scientific Databases. In *Proc. of the 28th Intl. Conf. on Very Large DataBases (VLDB)*, Hong Kong,China, Aug 2002.

[19] Sun Microsystems. The Java Security Model. http://java.sun.com/.

[20] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *1999 Intl. Conf. on Software Engineering*, pages 107–119, Los Angeles, CA, USA, 1999.

[21] E. Truyen, W. Joosen, and P. Verbaeten. Consistency Management in the presence of Simultaneous Client-Specific Views. In *Intl. Conf. on Software Maintenance*, Montreal, Canada, 2002.

[22] Xerox Corporation. The AspectJ Programming Guide. Online Documentation, 2002. http://www.aspectj.org/.