Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 238 (2009) 35–45

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Optimizing Aspectual Execution Mechanisms for Embedded Applications

Christine Hundt and Sabine Glesner

Institute for Software Technology and Theoretical Computer Science
Technical University of Berlin, D 10587 Berlin, Germany
Email: {*resix*|*glesner*}*@cs.tu-berlin.de*

Abstract

Applications for small embedded mobile devices are becoming larger and more complex resulting in the need for advanced modularization mechanisms. Aspect-oriented modularization is a promising approach to solve this problem, but the overhead of the existing programming languages restricts their practicability for small devices with limited resources. In this paper, we identify opportunities to optimize the aspectual execution mechanism at different levels within the virtual machine. First experiments show that these optimizations improve the execution time considerably, thus proving that with adequate optimizations, we enable the use of advanced aspect-oriented modularization techniques for small device Java applications.

Keywords: aspect-orientation, embedded systems, mobile devices, Java virtual machine, optimization

1 Introduction

During the last years, embedded mobile devices like hand-helds and mobile phones have become much more popular and have gained increasing computational power. This development has also stimulated the demand for more and more complex custom applications to be run on these devices. At the same time, such mobile devices are ruled by mass-market laws. This means that time to market as well as price per unit are critical success factors. To keep time to market small, software engineering methods are necessary that allow for reuse, adaptability and extensibility of software systems, rendering programming languages like assembler or C as no longer sufficient for this growing complexity. As in the area of desktop applications, advanced concepts for abstraction and modularization as e.g. object- or aspect-orientation are needed. At the same time, the hardware of such embedded devices will always be limited compared to desktop PCs, not least to keep their price per unit small. Hence, many software engineering methods are not directly applicable because they require too much memory space and/or computation time. This conflict between hardware restrictions and the need for more advanced software engineering concepts is characteristic for embedded systems.

In this work, we address the problem of developing software for applications running on embedded mobile devices. We require our solution to allow for better *maintainability*, *extensibility* and *reusability* of the embedded software. Moreover, our method shall be able to cope with an increased *variability* among the various static and dynamic versions of an application. Static variability arises for example when the same application is running on similar, yet different platforms of a product line. Dynamic variability comes up when the device changes its context and the application running on it needs to adapt to new boundary conditions. Finally, our method is required to be *efficient* and to exploit the underlying hardware at the best possible in order to meet the tight constraints of embedded hardware.

Our approach uses aspect-oriented programming and is based on the principal idea to modularize software adequately to achieve the above mentioned goals of maintainability, extensibility and reusability. Aspect-orientation is an extension of the object-oriented programming paradigm that allows for a better *separation of concerns*, i.e., for the separate implementation of kernel functionality from system-specific concerns representing e.g. configuration decisions that may come up at many places in a program. A subsequent weaving process combines functional and aspect code. These advantages of aspect-orientation do not come for free. One problem of this approach is the size of the resulting program code, especially for static weaving. It is often too large for embedded systems with restricted memory. For example, the *Java Micro Edition (J2ME)*, which is a Java platform for embedded consumer products, defines the *Connected Limited Device Configuration (CLDC)* with minimal assumptions for mobile devices such as cell phones. The CLDC assumes 160 KB of memory for storing the code that executes the Java virtual machine (JVM) and another 32 KB of memory used by the JVM during execution. These numbers clearly show the need for efficient use of memory. Another problem results from the demand for dynamic weaving, when aspects need to be added at run-time, e.g. to update software or to adapt software to changing contexts.

We solve this efficiency problem by extending the execution mechanism for aspect-oriented programs. Typically, aspect-oriented programs are executed by mapping aspect-oriented constructs to Java bytecode which is then executed on a standard Java virtual machine. Recent approaches [6,7] investigate the extension of the executing Java virtual machine by aspect-oriented features. Concerning the optimization potential of these two approaches, it turns out that the first solution offers only restricted optimization gains because it is restrained entirely on the limited operations of the Java bytecode. In contrast, the second solution offers much better optimization gains. Supporting aspect dispatch mechanisms at the level of the JVM is more promising as this allows for efficient weaving policies tailored to the specific needs of static and dynamic weaving in systems with only limited memory and computation power.

We have implemented our approach in the run-time environment of the aspect-oriented programming language ObjectTeams/Java (OT/J) [8]. OT/J is an extension of the Java programming language, performing the weaving of aspects at class loading time. The aspect dispatch logic is entirely realized at the level of Java bytecode, which is eventually executed by a Java virtual machine. In our work presented in this paper, we are extending the JamVM virtual machine such that different parts of the aspect execution mechanism for OT/J are optimized. First experimental results show that we significantly reduce the execution time with our approach and, hence, improve run-time performance.

This paper is structured as follows: Section 2 gives an overview of the foundations of aspect-oriented programming. In Section 3, we present our optimizations of aspect execution. In Section 4, we explain our implementation of selected optimizations in the existing execution environment of OT/J together with first experimental results. In Section 5, we discuss related work and in Section 6, we conclude and outline future work.

2 Foundations of this Work

2.1 Aspect-oriented Programming

Aspect-oriented programming (AOP) provides for enhanced *separation of concerns*. It facilitates a modularization of *crosscutting concerns*, which would be scattered across the module structure in purely object-oriented designs. The core functionality is implemented in the base program, while the crosscutting concerns are defined in separate *aspect* modules. These aspects define the crosscutting functionality together with *aspect bindings*, specifying the points in the execution of the base program (*join points*) at which they should be executed.

There is no closed definition of aspect-orientation, resulting in a large variety of aspect-oriented programing languages. Several of them allow for dynamic *activation* (and deactivation) of aspects. This also applies to *ObjectTeams/Java* (OT/J) [8], which supports advanced modularization concepts. In OT/J, aspect functionality is defined in *role* classes which adapt individual base classes. Roles are contained in *team* classes, which define a collaboration context for them.

2.2 Aspect Weaving

The aspect code has to be *woven* into the base code at some point in time so that it can be executed at the defined join points of the base program. There are different approaches to accomplish aspect weaving: *Compile-time weaving* statically weaves the aspect functionality into the base code. *Run-time weaving* can be performed at class loading time, before the base classes are executed by the JVM, or even later during the execution of the program. This can be done by a run-time component outside the JVM or inside the JVM itself. The later the aspect weaving is performed, the more dynamically the aspects can be added to the base program allowing for more flexible context-aware adaption of the executed applications. Unfortunately, dynamic aspect weaving suffers from more effort at run-time. Static weaving, on the other hand, increases the code size, which is also critical for small devices.

Currently, OT/J performs load-time bytecode transformation to weave the aspects into the base code. In this paper, we investigate how the execution time of such woven program code can be optimized by enhancing the functionality of the executing JVM. This pays off by a significant gain in execution time, as we have shown in our first experiments described in Section 4. And moreover, it paves the road for our long-term goal, namely the dynamic weaving of aspects at run-time, instead of at class loading time, in order to also further optimize the code size of the executed applications.

3 Optimizing Aspect Execution

Purely additive realizations of aspect-oriented programming languages per se produce a certain amount of overhead, compared to the programming language they extend. In this section, we start by illustrating the overhead caused by the different tasks performed during aspect execution. In the succeeding subsection, we present our optimizations for the different kinds of overhead. The subsections in 3.2 propose optimizations for the overhead identified in the corresponding subsections in 3.1.

3.1 Sources of Overhead

The overhead of aspect-oriented program execution is caused by additional control flow using additional data structures. Furthermore, the overhead of the weaving process itself has to be considered, especially for languages that apply post-compile time weaving, like OT/J. One could argue that the execution time is more relevant because weaving is done only at an initial phase. Nevertheless, dynamic class loading blurs the border between these two phases. Thus the overhead of the weaving process can also be critical during program execution. Taking this consideration into account, we can classify the sources of overhead for the language OT/J by the following categories.

3.1.1 Aspect Registration/Activation

Aspects have to be linked in some way to the base classes they adapt. This information is needed when looking up the adapting aspect instances while executing a base method. For this some kind of registration mechanism is needed, usually realized as list data structure associated to the adapted base class. If furthermore dynamic activation/deactivation of aspects (teams) is supported by the language, then these data structures have to be dynamically updated during run-time. OT/J supports different activation policies. Aspects can be activated explicitly/implicitly and globally/thread-locally (see [8], §5). Naive realizations of these policies can cause significant overhead. Thread-local activation, for example, makes the execution of an aspect dependent on the current thread context. On the level of Java bytecode, there are no efficient mechanisms for this purpose. Using the Java class `ThreadLocal` is a possible, but unsatisfying approach.

3.1.2 Aspect Dispatch

The execution of aspectual behavior is typically realized by additional calls to aspect instances. Dispatching from a base method to an adapting aspect method adds additional dynamic method lookup(s). This is necessary if the aspect language supports polymorphic method overriding also at the aspect side. Dynamic method dispatch is always expensive and should be avoided due to performance reasons, if possible. The identification of avoidable dynamic dispatch constitutes good potential for possible optimizations. The execution of aspectual code is subject to special conditions, which does not hold for method dispatch in general. Possible optimizations could exploit these conditions.

3.1.3 Run-time Infrastructure

In the face of dynamic class loading, a post-compile time weaving process is always incremental. Therefore information about aspects, adapted base methods, inheritance structures etc. has to be maintained during run-time. This necessitates additional data structures and causes extra overhead. In OT/J, for example, the base-side aspect inheritance requires additional run-time information. Aspects adapting a base class do also affect subclasses. If affected methods are overwritten, also the subclasses have to be woven accordingly. In a dynamic weaving setting, the subclasses could be already loaded by the virtual machine. To locate them, some kind of *subclass lookup* is necessary. In the current version of OT/J, run-time information needed for the weaving process are stored by means of Java data structures.

3.1.4 Advanced Modularization Mechanisms

OT/J features inheritance on the level of collaborations (teams). For the roles inside a collaboration this results in a special inheritance relation to their counterpart in a super collaboration, called *implicit inheritance*, cf. [8], §1.3.1. Currently, this mechanism is realized by a copy of the 'inherited' bytecode and the generation of additional interfaces. This increases the code size and necessitates the use of less efficient interface method invocation mechanisms. (Note that the bytecode instruction `invokeinterface` is less efficient than `invokevirtual`.)

3.2 Optimization of the Run-time Environment

As we have discussed in the previous section, there are different kinds of overhead affecting the efficiency of aspect-oriented program execution. These overheads can be reduced much better at the level of the JVM than at the level of Java bytecode because the JVM maintains data structures necessary to support method execution and is responsible for the execution (interpretation) of bytecode instructions. We propose the following extensions for the support of aspect execution:

- We propose additional *data structures* that are introduced to store additional information useful for the execution of aspectual code (e.g. aspect registration, implicit inheritance information, role caching).
- We propose additional (or adapted) *bytecode instructions* that are used to support optimized aspect execution. These instructions will make use of the introduced data structures.

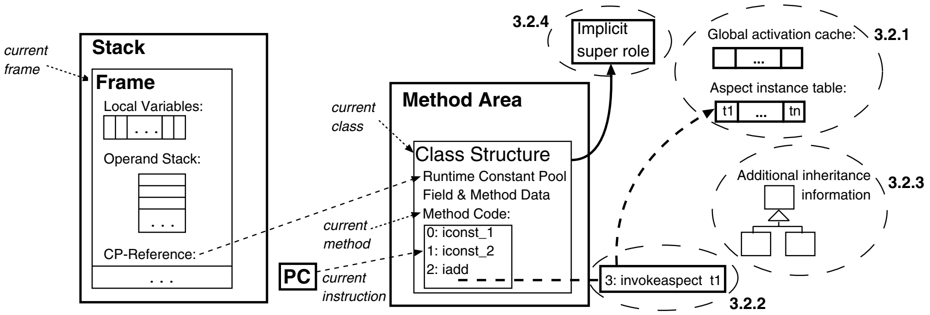

Fig. 1. JVM Architecture with Optimizations

- Finally, we will provide an *interface* (API) to allow the weaving process to access information stored in the JVM data structures (e.g. from the run-time infrastructure).

Concrete optimizations are detailed in the following and illustrated in Figure 1.

3.2.1 Team Activation Infrastructure

In the current OT/J implementation, the infrastructure necessary for team (aspect) activation is added to every adapted base class. Arrays for storing aspect instances and methods to access these data structures (`addTeam(Team)` and `removeTeam(Team)`) are added to the corresponding class files. These methods are called by the de-/activation methods of the teams.

Modeled on the *advice instance tables* of [7], we propose to move the aspect registration mechanism to the JVM-level, allowing for various optimizations. First, we can move the data structures to the VM-internal data structures representing classes. To allow access to these data during aspect activation, we need to extend the interface of the VM. In a second step, we introduce a global cache as optimization for consecutive activation and deactivation of the same aspect instance. This is a common pattern for implicit team activation. Note that implicit team activation guarantees a coherent aspectual control flow if a public method of a team or a role is called, cf. [8], §5.3. Next, the aspect storage could be populated in a more context specific way. Currently, aspect instances are stored per class, but during aspect execution only a subset actually adapts the current base method. Moreover, in the case of thread-local activation, this subset is further restricted by the currently executed thread. At VM-level, the thread context is used for several purposes. Thus it should be more easily accessible also for thread-local team activation mechanisms.

3.2.2 Aspect Dispatch

We propose to develop a specialized aspect dispatch mechanism working on the VM-internal activation infrastructures introduced in 3.2.1. On the bytecode-level, this mechanism can be addressed via a new bytecode instruction (`invokeaspect`). Thus, the current wrapper-based approach can be refined. Furthermore, we plan to introduce a special (non-dynamic) method lookup calling the original base method during execution of replacing aspect functionality (*base calls*). This is possible because the original method is known exactly as it has been called initially.

3.2.3 Aspect Inheritance

Like for other run-time information, VM-internal data structures can be used to store additional inheritance information. We plan to realize the subclass lookup necessary for aspect inheritance VM-internally, using this information.

3.2.4 Implicit Role Inheritance

Roles of a team implicitly inherit from roles with the same name in a super team. This inheritance relationship is no normal type inheritance and is only valid in the context of a surrounding team instance. We plan to add a reference to the implicit super role for role classes in the VM, analogously to the super class reference stored in class structures, avoiding code copying and additional interfaces.

4 Case Study and Results

We have implemented parts of the optimizations introduced in Section 3 in a real virtual machine to measure their benefit.

The Object Teams Run-time Environment (OTRE) is currently responsible for aspect weaving in the OT/J programming language. As described earlier, the base code is transformed at class loading time and subsequently executed by the JVM. The current OTRE works together with any standard JVM. Typically, the Sun JVM is used, which is not suitable for small devices because it is too large. We chose an adequate JVM implementation (cf. Sec. 4.1) which we connected to the OTRE. This configuration provides a basis for our optimizations to be implemented in the VM. This approach facilitates a direct comparison with the original OT/J implementation, allowing a precise measurement of the performance improvement. Furthermore, we can add our optimizations incrementally, while at any time sustaining the full functional range of the OT/J language for benchmarking.

In this section, we motivate our choice of a VM implementation, outline our implementation and report our first experimental results.

4.1 VM Implementation

The selected VM implementation had to fullfill a set of criteria: availability (open-source), extensibility, performance, target architectures and compatibility to the existing OT/J. We have selected the JamVM [1] for implementing our optimizations. It supports the full JVM specification version 2, although it is extremely small and applicable for embedded devices. For example, the JamVM has been ported to iPAQ 3950 and Neo1973. The JamVM is implemented in C, with a small amount of platform dependent assembler code.

Other candidates we considered were the KVM [3] and the phoneME [2] VM. In contrast to the JamVM, both only support Java ME. The KVM does not seem to be maintained during the last time. phoneME features a JIT compiler, but this makes its implementation much more complex than the JamVM and not suitable for prototypically testing optimizations.

4.2 Prototypical Implementation of Optimizations for Team Activation

Previous performance measurements [5] identified (implicit) team activation as a significant element for the overhead of OT/J program execution. Hence, as first optimization point, we have chosen the team activation mechanism, realizing two initial stages as described in Sec. 3.2.1.

Team Activation Infrastructure at VM-Level

In a first step, we have moved the team registration mechanism to the VM-level. The data structure representing a class inside the JamVM was extended by a reference to the newly introduced registration data structure. In contrast to the previous weaving strategy of OT/J, now **every** class has this structure. Initially, this causes a very small overhead (20 bytes on x86 architecture), and only if a class is actually adapted by an aspect (team) advanced initialization and memory allocation take place.

As before, an interface for adding and removing team instances to a base class is needed. Additional functionality to access the array of active team instances is necessary because the remaining aspect execution mechanism resides at bytecode-level at this stage of optimization. We were able to integrate this functionality into the interface of the JamVM very conveniently. The JamVM provides internal implementation of Java mechanisms like class loading and reflection. At the Java-side, a class with native method declarations serves as interface. Calls to internal native methods are not executed via JNI (Java Native Interface), but directly implemented in the VM code. This approach was transfered to the team registration mechanism. Finally, we have adapted the weaving strategy of the OTRE to use the new VM-internal mechanism.

Global Cache for Implicit Team Activation

Next, we implemented a global cache for team activation. This cache stores the previous state of the activation structure when a new team instance is added to a base class. If the same team instance is removed again, before any other 'add' or 'remove' operation is performed, the cache is written back to the corresponding activation structure. As argued in 3.2.1, such a consecutive activation and deactivation of the same team instance is a typical execution pattern in the event of implicit team activation.

4.3 First Results

We have measured the benefit of our implemented optimization with the following benchmark, cf. Figure 2. A team-level method was executed 1000 times, causing 1000 implicit team activations and subsequent deactivations. We have varied the number of previously activated team instances from 0 to 10000 as shown along the

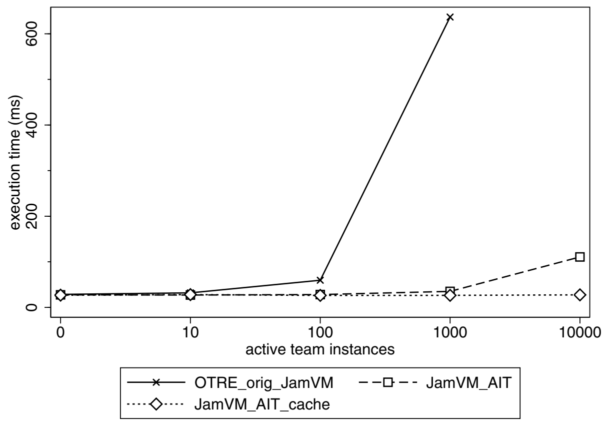

Fig. 2. Benchmarks

Table 1
Execution Times in ms for the Optimizations

	ati_0	ati_10	ati_100	ati_1000	ati_10000
OTRE_orig_JamVM	28.55	31.93	59.56	636.79	3720.37
JamVM_AIT	27.58	27.14	28.17	35.12	110.67
JamVM_AIT_cache	26.86	28.03	26.33	26.47	27.50

x-axis in the diagram. The y-axis represents the consumed time in milliseconds when run on Intel Pentium processor with 1.60GHz.

With this benchmark, we compare the performance of the original JamVM (OTRE_orig_JamVM) with the optimization which moves the activation infrastructure to the VM (JamVM_AIT) and the version with an additional global cache (JamVM_AIT_cache). The results, completely shown in Table 1, are the means of 100 values for each measurement. The difference of the two optimizations and the OTRE_orig_JamVM is statistically significant (0.05 level) except for 0 active team instances (ati_0). For 1000 and more instances a significance is given also for the advantage of JamVM_AIT_cache compared to JamVM_AIT.

Our experimental results show that JamVM_AIT yields a considerable performance gain compared to OTRE_orig_JamVM. The execution time is reduced by a factor of 2 for 100 instances and even more, the more team instances have been activated in advance. The gain of JamVM_AIT_cache is even larger, showing a nearly constant execution time. When the cache is used, no team instances have to be moved in memory, which had caused the growing complexity for the other configurations.

With our experiments, we have shown that the time efficiency of aspect activation can be improved significantly. By implementing also the other optimizations that we have proposed in Section 3.2, we expect to further improve the execution of aspect-oriented embedded applications.

5 Related Work

Several research has been done in the field of optimizations for AOP in general. Steamloom [6] is an extension to the Jikes Research Virtual Machine with direct support for general AOP language mechanisms. The introduced Advice Instance Tables (AIT) [7] with its efficient lookup mechanism are related to our team activation mechanism, but this work does not target embedded small applications. Approaches like [4] aim at optimizing aspect dispatch by replacing dynamic aspect method lookup by static method calls. This does not solve our problem because the dynamic method lookup of aspect methods is important for an adequate modularization of developed software. Aspect C++ [9] is also intended for the generation of small and efficient code for constrained environments but does not focus on reusability of the aspects and uses a purely static weaving mechanism.

6 Conclusion and Future Work

As embedded software grows significantly, it becomes more and more important to apply software engineering methods in this area. In this paper, we have proposed optimizations of the aspect execution mechanism on the VM-level aiming for an applicability of AOP for embedded mobile devices. Our first results show a considerable performance gain for the aspect activation mechanism of OT/J. So far, the presented benchmarks did not involve actual aspect execution. In future work, we plan to implement more optimizations, particularly an aspect dispatch mechanism working on the VM-internal activation infrastructures, and to measure and to evaluate the overall benefit. We also plan to investigate hybrid approaches that combine compile-time and run-time weaving, e.g. by weaving large code dynamically and small code statically.

Furthermore, we also plan to take optimizations at application-code level into account. This could result in (maybe speculative) optimizations which utilize certain constraints on the execution of OT programs, which are not necessarily valid for equivalent bytecode. Examples are the repeated use of certain values (e.g. role instance caching) or an efficient (non-dynamic) method call for base calls. Such optimizations may demand additional analyzes at compile time.

Finally, we want to validate our approach on a real embedded device with a case study. We plan to do this with the Linux-based open source smart phone Neo 1973 and an OT/J implementation of a game product line.

Acknowledgement

This work has been funded by the German Science Foundation (DFG) under the Emmy Noether Grant No. GL 360/1-2. We also thank our student Magdalena Luniak for her assistance regarding the implementation of the optimization and the analysis of the measurement.

References

[1] JamVM homepage: http://jamvm.sourceforge.net.

[2] phoneme homepage: https://phoneme.dev.java.net.

[3] White Paper on KVM and CLDC: http://java.sun.com/products/cldc/wp/.

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

[5] Paul Häder. Benchmarking und Optimierung der Aspektwebestrategie von ObjectTeams/Java. Master's thesis, Technische Universität Berlin, 2006.

[6] M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Software Technology Group, Darmstadt University of Technology, 2006.

[7] M. Haupt and M. Mezini. Virtual Machine Support for Aspects with Advice Instance Tables. *L'Objet*, 11(3):9–30, 2005.

[8] Stephan Herrmann, Christine Hundt, and Marco Mosconi. ObjectTeams/Java Language Definition — version 1.0. Technical Report 2007/03, Fak. IV, Technical University Berlin, 2007.

[9] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in AOP with AspectC++. In *New Trends in Software Methodologies Tools and Techniques*, Frontiers in Artificial Intelligence and Applications, pages 33–53, 2005.