Disentangling Virtual Machine Architecture

Michael Haupt¹, Celina Gibbs², and Yvonne Coady²

¹ Software Architecture Group
Hasso Plattner Institute for Software Systems Engineering
University of Potsdam, Germany
² University of Victoria, Canada
michael.haupt@hpi.uni-potsdam.de, celinag@uvic.ca, ycoady@cs.uvic.ca

Abstract. Virtual machine implementations are made up of intricately intertwined subsystems, coordinating largely through implicit dependencies. An analysis of several virtual machine implementations reveals the presence of crosscutting concerns—concerns that cannot be modularized through traditional means and whose implementation is scattered over and tangled with the source code of other modules. This paper proposes an architecture approach for virtual machines. The approach regards a virtual machine as a conglomerate of service modules coordinated through explicit interfaces using linguistic mechanisms based on aspect-oriented programming techniques.

1 Introduction

Virtual machines (VMs) consist of subsystems with clearly defined responsibilities. Unfortunately however, said subsystems are not clearly separated from each other, e.g., implemented as modules with clean boundaries that exchange information via dedicated explicit interfaces. Instead, relations among and dependencies between subsystems are represented implicitly in the implementation. Hence, service coordination is unclear, and adaptation is at best ad hoc.

As a consequence, VM architecture can be cleanly described only at a very high level of abstraction. As soon as actual implementation decisions come into play, it becomes virtually impossible to reason about a single VM subsystem in terms of its functionality and interface—instead, the way other subsystems interact with it must be considered. This results in assumptions about other subsystems, and the resulting coordination, being hardwired into the system.

Coordination between VM components often can be described at a high level of abstraction in terms of constructions like “when this happens in module X, module Y must react that way”. In fact, research has shown promise in mapping component interaction at the implementation level to the higher level of requirements and design [11]. But, given that coordination such as this is often implied, it should lend itself to be explicitly represented in the system. However, at lower levels of abstraction, i.e., in code, representations of such interactions are often implicit (cf. above), and assumptions about them are hidden in code. Moreover, the nature of coordination of this kind is often asynchronous: e.g.,
the decision to subject a method to optimized (re)compilation is drawn by the adaptive optimization subsystem, but it is not carried out immediately by the compilation subsystem until ample time is available. To express asynchronous interactions, queues and the like have to be implemented and maintained.

We believe that VM implementations contain many crosscutting concerns [12] that represent coordination between subsystems. AOP techniques have been shown to provide support for coordination between software components [8]. We argue that these AOP techniques can be used to improve the ways in which the architectures of virtual machines are expressed, and further that extensions to these techniques would be appropriate within this domain. In brief, the proposed architectural view is that of a VM being a conglomerate of services provided to an application. Each of the services constitutes a module with a clear interface. The interface consists of a set of operations that the module can be asked to perform—its API—, and of another set of signals that it can expose when certain internal properties require it—its XPI (crosscut programming interface) [10]. The circumstances under which these signals are raised are to be internally described using aspect-oriented means. The coordination between service modules is to be achieved declaratively by aspect-oriented means as well.

2 Decomposition and Crosscutting in Virtual Machines

We have considered eight modern VMs, both C- and Java-based, and identified crosscutting concerns in their architectural decomposition, some of which we will describe in this section. We briefly describe our experiences with extending existing VM infrastructures using AspectJ, and problems encountered due to the general lack of structural support within this intricate domain.

Due to the tangled nature of VM concerns, it is often difficult to separate some concerns using AOP. For example, the GCspy [14] heap visualization framework is designed to visualize a wide variety of memory management systems. A system as complex as a VM benefits greatly from non-invasive, pluggable tools providing system visualization while minimizing the effects on that system. Such tools inherently have many fine-grained interaction points that span the system they are visualizing, lending themselves to an aspect-oriented implementation. GCSpy functionality involves (1) gathering data before and after garbage collection, and (2) connecting a GC Spy server and client-GUI for heap visualization. The resulting AspectJ code touches 12 classes in the Jikes RVM, has a 1:1 ratio of pointcuts to advice, and uses a collection that spans before/after/around advice and can be further characterized as a heterogeneous concern [5, 2].

Similarly, modifying the OpenVM to support Software Transactional Memory (STM) involved a series of changes that lent themselves to an aspect-oriented implementation. The resulting AspectJ code touches 13 classes, has a 1:1 ratio of pointcuts to advice, and heavily employs non-proceeding around advice.

In previous work, it has been demonstrated how the Jikes RVM’s modularity can be enhanced even with a naïve implementation of aspects, and how these aspects impact system evolution [9]. Now, we consider a more qualitative assess-
ment of the representations of the aspects themselves, and the ways in which AspectJ [13, 3] could be augmented to better support the needs of crosscutting concerns in VMs.

**Join Points in VMs** It is our experience that the types of join points—both static and dynamic—exposed by most existing join point models are not sufficient for expressing the special needs of crosscutting concern composition in VM implementations. A join point model fit for this implementation domain should still exhibit the features of, e.g., the AspectJ join point model [13] with several extensions, such as stateful aspects or tracematches [7, 1].

**Pointcut Descriptors** VM subsystems frequently invoke other subsystems when certain “points of interest” occur. Normally, such interactions inherently require access to dynamic, often shared, VM system state. The ability to construct stateful aspects [7], and express them using the appropriate means, e.g., tracematches [7, 1], is critical in this domain.

**Advice** Our survey indicates the need for some concerns to interact in a detached way; utilizing asynchronous advice [4], as met in the AWED language [6]. Asynchronous advice cannot be expressed directly using simple AspectJ mechanisms, as the required queues and associated state have to be introduced as explicit data structures. Though there is no such concept as an asynchronous advice in traditional AOP advice models, we believe this to be highly desirable in VMs. Given that there are very likely many VM subsystems that do not interact synchronously, modeling such services as crosscutting concerns calls for providing a mechanism allowing for such definitions.

### 3 Disentangled Virtual Machine Architecture

We propose an approach for virtual machine architecture that treats the various subsystems of a VM implementation as *services* offered to an application run by the VM. Services are modules, and they have clean boundaries.

For illustration purposes, we consider the structure of traditional VM subsystems seen from at a high level of abstraction as depicted in Fig. 1. Each of the shapes in Fig. 1 represents the source code of one classic subsystem. The different subsystems are mostly modularized, but parts of their implementations are scattered over the system and tangled with code pertaining to other subsystems.

Our approach to VM architecture focuses on establishing a clear modularization for VM subsystems. This is achieved by applying aspect-oriented programming techniques. The different subsystems of a VM are regarded as *services* that the VM provides to the running application—e.g., services for application representation, scheduling, memory management, etc.

A service’s boundary is defined in terms of an interface that, on the one hand, provides means to invoke functionality of the service (its API). On the other hand, the interface exposes certain points of interest that occur internally and may be of interest to other services (the service’s XPI [10]).
This principle is illustrated in Fig. 2(a). A service module is a superstructure comprising of several actual implementation modules (e.g., classes) and have a large internal complexity, but it is a module at an architectural level; one that can clearly be assigned a responsibility in terms of VM functionality.

During the execution of service functionality, of course, certain situations arise where service interaction takes place. Applying the usual manner of VM implementation, service interaction code—e.g., through invocations of other services’ functionality—would be hardwired into the source code of the service.

The architecture approach we propose takes a different road: the situations where service interaction should take place are declaratively described using pointcuts that quantify over the join points occurring during the execution of service functionality. Such pointcuts are however not directly associated with advice, but instead their matching constitutes the occurrence of a point of interest exposed from the service module (denoted by bold dashed arrows).
The join point model and pointcut language applied at this level of granularity can be any that are sufficiently powerful to express the aforementioned situations of interest. An exposed point of interest comes with the according context information to be exploited by client services.

Actual VM architectures are declaratively described. The static structure of the implementation is described by choosing a set of concrete services; and the dynamic service interactions are realized by declaratively describing them in terms of reactions to points of interest exposed from service modules—at the level of service composition, the join point model is constituted by all such exposed points of interest. Said points are called service-exposed join points.

Fig. 2(b) shows, at the same level of abstraction as Fig. 1, what VM architecture looks like when the principles described above are applied. The VM consists of a collection of clearly bounded service modules. Interactions among them are described in terms of service-exposed join points, pointcuts quantifying over them, and invocations of service functionality in case such a pointcut matches.

In summary, aspect-oriented programming techniques are applied in this model at two degrees of abstraction. On the one hand, intra-module execution-level join points are quantified over to make up service-exposed join points. On the other, service-exposed join points are quantified over to drive service interaction. The two join point models found in the architecture approach are separate from each other. Service-exposed join points can be assigned meaningful names, improving declarativeness at the architectural level.

4 Summary

We have presented an analysis of crosscutting concerns in VM implementations and given examples for them. Based on the observation that such systems contain many intricately combined crosscutting concerns, we have proposed an approach to organizing VM architectures based on service modules, i.e., modules constituting well-defined services that the VM provides to applications run by it.

Coordination between service modules is organized using service-exposed join points. They are defined using aspect-oriented means. The coordination of module interaction is also done via aspect-oriented programming techniques, allowing for the expression of VM architectures by means of explicit dependencies.

The following issues have to be dealt with to advance this research:

– Typical service modules. The aforementioned set of VM service modules has been derived from an analysis of several concrete implementations. For these modules, “typical” interfaces have to be identified, i.e., it has to be determined which particular services can be asked from each service module, and how control over these services from the outside can be designed without sacrificing information hiding. Moreover, more VMs have to be taken into account to gain a more complete model of the domain.

– Interaction patterns. To be eventually able to formulate clear architectural guidelines and programming language facilities to transcribe them to software, the different types of interactions between service modules need to
be identified. Their characterisation will most likely impact the way linguistic means for expressing service module interaction in terms of pointcuts quantifying over service-exposed join points and advice execution in terms of service module invocations are designed.

- **Language facilities for service modules and join point exposure.** Current aspect-oriented programming languages mostly support aspects as modules, but the notion of “interface” is often limited in that join point exposure is not supported. Instead, aspects have almost arbitrary control over modules, breaking core modularity principles. We favour to reason about service modules and their interactions solely in terms of their interfaces.

- **Design of VMADL.** Service modules exposing join points require coordination in order to form an actual virtual machine implementation. To the end of coordinating such modules, a dedicated domain-specific architecture description language is to be devised that supports the expression of VM implementations in terms of service modules.

**References**