Disentangling Virtual Machine Architecture

Postprint*

published in IET Software, volume 3, issue 3 (June 2009), pages 201–218

Michael Haupt[1], Bram Adams[2], Stijn Timbermont[3],
Celina Gibbs[4], Yvonne Coady[4], Robert Hirschfeld[1]

[1]Hasso-Plattner-Institut, University of Potsdam, Germany

[2]GH-SEL, Ghent University, Belgium

[3]Programming Technology Lab, Vrije Universiteit Brussel, Belgium

[4]University of Victoria, Canada

{michael.haupt;hirschfeld}@hpi.uni-potsdam.de, bram.adams@ugent.be,
stimberm@vub.ac.be, celinag@uvic.ca, ycoady@cs.uvic.ca

Virtual machine (VM) implementations are made of intricately intertwined subsystems, interacting largely through implicit dependencies. As the degree of crosscutting present in VMs is very high, VM implementations exhibit significant internal complexity. This paper proposes an architecture approach for VMs that regards a VM as a composite of service modules coordinated through explicit bidirectional interfaces. Aspect-oriented programming techniques are used to establish these interfaces, to coordinate module interaction, and to declaratively express concrete VM architectures. A VM architecture description language is presented in a case study, illustrating the application of the proposed architectural principles.

1 Introduction

Virtual machines (VMs) are inherently complex systems. They normally consist of several subsystems, such as an emulation engine, a memory manager, and so forth [1]. Each of these subsystems bears a certain inner complexity: for example, the emulation

*This paper is a postprint of a paper submitted to and accepted for publication in *IET Software*, and is subject to Institution of Engineering and Technology copyright (www.ietdl.org/journals/doc/IEEDRL-home/copyright.jsp). The copy of record is available at IET Digital Library www.ietdl.org.

engine may consist only of an interpreter, or of a combination of the same with a just-in-time (JIT) compiler for optimised execution. Such a combination brings about an adaptive optimisation subsystem that monitors application execution and analyses it to draw optimisation decisions.

All of these subsystems are intertwined in subtle ways even in simple VM implementations. The choice of a garbage collector frequently influences object layout and JIT compiler code generation, e. g., when garbage collection maps have to be built. The choice of mechanisms for synchronisation and locking also influences the two, because object (header) layout is affected based on that choice.

In other words, even though VMs consist of subsystems with clearly defined responsibilities, these subsystems are not clearly separated from each other: they are not 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.

As a consequence, VM architectures can be described only at a very high level of abstraction. As soon as actual implementation decisions come into play, it becomes very hard to reason about a single VM subsystem in terms of its functionality and interface—instead, the ways other subsystems interact with it must be considered. This results in assumptions about other subsystems being scattered over and tangled in multiple locations.

From a design perspective, even *design decisions* are hardwired. For instance, the selection of a memory management scheme influences many different parts of a VM implementation, rather than just the memory management subsystem. In settings where VMs are regarded as product lines that should support different forms of their various subsystems—e. g., the Jikes Research VM [2, 3]—, or that should be deployable on different platforms—e. g., Sun's HotSpot Java VM [4]—it must be possible to create and handle abstractions for dealing with changing subsystem interactions. It may even be desirable to support *dynamic* variability, e. g., by dynamically adapting memory management strategies in response to the running applications' requirements [5]. Hardwiring such decisions clearly reduces variability support.

Interactions between VM components can often 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". At lower levels of abstraction, i. e., in code, representations of such interactions are mostly implicit, and assumptions about them are hidden.

For instance, a reference counting garbage collector will have to adjust reference counter values whenever reference assignment takes place. The execution of the simple assignment `x := y` involves no less than three VM subsystems or building blocks: when the interpreter (1a) executes the statement, or when the JIT compiler (1b) generates code for it, it must be ensured that the garbage collector (2) increments and decrements reference counts, which in turn are stored in a given location specified by object layout definitions (3). This example also shows that both static and dynamic properties of the VM source code are involved: the effects of garbage collector choice on object layout are static, while interactions between execution logic and garbage collector are dynamic.

The observed issues in VM implementations bear some notable similarities with *crosscutting concerns*, which are also *scattered over* and *tangled with* other concerns' implementations. Moreover, crosscutting concerns can exhibit effects on both static and dynamic properties of code. We therefore argue that aspect-oriented programming (AOP) techniques can be used to address the aforementioned issues.

VM implementations usually take place at a level of abstraction where copious control over the underlying machine is required to gain consistent behaviour and predictable performance. That is, software concerns that are, in typical application development scenarios, usually regarded as non-functional ones, e. g., memory management and concurrency, are core *functional* concerns in the VM domain. As a consequence, some trade-offs have to be taken into account to apply AOP at full potential in this domain, mostly related to the treatment of AOP infrastructure, and to the degree of control that a programmer has over the infrastructure.

AOP research has evolved in several directions, one of the most proliferating of which is pointcut language design. The first mature pointcut languages could quantify over the execution of applications in terms of method call and execution, field access, exception handler execution and the like [6]. Soon after, pointcut languages could quantify over control flow. More recent developments have explored the use of logic-based pointcut languages [7] and added capabilities to let a pointcut match when certain join points occur in a given order over a period of time [8, 9, 10]. Another extension allows for quantifying over interconnections between objects on the heap [11].

To express the intricate subsystem relationships met in VM implementations, such semantically rich pointcuts are clearly desirable. However, their implementations typically entail the extensive use of residual code [12] that, depending on its implementation, may have significant impacts on performance. In the VM area, performance is a core concern, and the "standard" implementations of residues may not be optimal in all cases. Therefore, we propose to give developers the ability to create *domain-specific abstractions* for VM implementations that allow for expressing intricate relationships between VM subsystems while retaining as much as control over the implementation of possibly required residual code. That is, domain-specific programming language support for VM implementations should support reasoning about VM modules and their interactions *declaratively* while not hindering the developer in their implementation.

A second research direction relevant to this paper is the degree of freedom in advising code. One end of the spectrum allows to match join points throughout the entire code base, even in terms of encapsulated state and behaviour (cf. privileged aspects in AspectJ). The opposite extreme restricts the join points which can be advised by explicitly exposing possible join points [13, 14].

To achieve both modular reasoning and more control in VM implementations, we propose an architectural approach that is centred on treating a VM as a composition of *services* provided to an application. Each of the services constitutes a module with a clear interface. The interface is *bidirectional*, offering a set of operations that the module can be asked to perform, and of a set of *exposed join points* that occur at run-time when certain internal properties are met. The circumstances leading to the occurrence of such joins point may be of great complexity, but the complexity can be reasoned about in

terms of domain-specific join point models constituted by join points exposed by service modules.

While this clearly resembles the crosscut programming interfaces (XPI) [13] or open modules [14] approaches, we refrain from modularising absolutely all features into services and connecting their parts through bidirectional interfaces. Our case study made it clear to us that a completely and cleanly modularised VM is close to impossible to achieve. Some concerns are just too crucial or fundamental, and are used throughout all services. As the number of such concerns is small, it is acceptable to allow some degree of tangling in them. Scattering should pose no problem, as the concerns in question tend to be localised well. One of the best examples for this is object layout. Garbage collection services need to know the exact layout and may even have to add extra fields. Hence, on the one hand, general-purpose aspect languages offer too much freedom (everything can be advised), while XPIs and open modules are too restricted in that they impose a very strong modularity-oriented point of view on code. We adopt a more pragmatic approach, trying to combine the advantages of both extremes.

The two principles of our approach—domain-specific abstractions and bidirectional interfaces—are, as a proof of concept, implemented in a declarative architecture description language called *Virtual Machine Architecture Description Language* (VMADL). VMADL is used to describe domain-specific join-point models in service module interfaces and to coordinate service module interaction, using aspect-oriented techniques.

This paper makes the following contributions:

- an analysis of service decomposition in existing VM implementations,
- a presentation of our architectural approach based on domain-specific abstractions, bidirectional interfaces, and pragmatic employment of AOP, and
- a case study covering garbage collection and multithreading services.

The next section presents an analysis of several VM implementations, focusing on both the variability found in traditional decompositions and the relationship between current infrastructures and VM services in general. In Sec. 3, the service-based VM architecture proposal is presented in depth. A case study in which these principles have been applied to a VM implementation is presented and evaluated in Sec. 4. Sec. 5 discusses related work, and Sec. 6 gives an overview of future work and concludes the paper.

2 An Analysis of Decomposition in Virtual Machines

This section surveys current strategies for decomposing VM infrastructure in source code. We consider eight modern VMs, both C- and Java-based, and identify the corresponding decompositions in terms of core services and directory/package structures. The survey reveals two key results: (1) current VMs display substantial variability in the nature of their decomposition, and (2) the internal structure of these systems does not explicitly reflect the architectural elements they all inherently share. This analysis supports the

argument that explicit composition of VM services could be best supported in a domain-specific fashion, where join point models specific to VMs in general could be formulated according to states in the execution of the system that are common to VMs. Further, we believe these domain specific join point models must allow module interaction to be specified relative to well-defined VM states, such as during and after system booting, involving the intialisation and startup of the VM.

The following three subsections provide a structural analysis based on a survey of eight high-level language VMs. The first subsection overviews the system structure conveyed by directory structure of the source tree along with the identification of core services across all VMs. The following subsection maps the representation of these services onto the top-level system modularity dictated by the directory structure. Concrete code examples in Sec. 2.2.1 illustrate the interaction between services that occurs at the level of directory and file modules.

2.1 System Decomposition

When surveying the fundamental abstractions that constitute VMs, it becomes clear that there are many ways to decompose the system into modules and subsystems. For example, Fig. 1 shows a spectrum of six different source trees for C-based VMs. The use of directory structures is one way of imparting knowledge of an architectural design from the resulting top-level modularisation. Although at a high level, this evaluation reveals the variety in dominant decomposition strategies and demonstrates that the set of services that constitute the core VM are not entirely obvious nor are they agreed upon between architectures.

For example, consider the part of the system responsible for the operating system (OS) interface. In Kaffe [15], OS variation is localised within the *systems* subdirectory; in Harmony [16], a subdirectory tree associated with each high-level VM subdirectory contains OS-specific code; in TinyVM [17], variation is contained within *vmsource*; in KVM [18], high-level subdirectories separate OS concerns within the system; in SableVM [19, 20], processor variation is contained within the *src* directory; and in LeJOS [21], a top-level platform directory accomplishes a similar structure, albeit for a single processor. While each of these systems has a different dominant decomposition, they all have a common goal: to achieve modularisation of core services in a way that will scale as the VM evolves. As this survey reveals, there is no single strategy emerging as a winner to provide a clear modular decomposition of all services into directory structures. The example of the OS interface demonstrates that all services cannot be cleanly modularised through traditional procedural or object-oriented separation mechanisms. Considering the memory management example, this is a critical and highly tuned service within a VM, but we see that only Kaffe decomposes it into a separate folder.

Navigating through the source trees and deeper into the files of each of these systems, we can begin to tease out core abstractions that are common to this subset of systems. Core to each is the functionality associated with representation, OS interface, scheduling, execution, and adaptive optimisation. By representation, we are referring to the internal infrastructure for application elements such as classes, methods and objects. The OS

```
1 public void run(Context ctx) throws PragmaInline {
2   Interpreter.run(processor_, ctx.nativeContextHandle_);
3   MemoryManager.the().observeContextSwitch(Interpreter.getContext(processor_));
4 }
```

Listing 1: The `run` method of the OVM.

interface refers to the code that sits directly on top of the underlying system. Scheduling is taken broadly to refer to threads and synchronisation mechanisms. Memory management, again taken broadly, refers to allocation and garbage collection. By execution we mean the actual run-time engine, the interpreter and JIT compiler for applications. Finally, by adaptive optimisation we are generally referring to run-time optimisations employed during application execution.

2.2 Service Representation

Given the conceptual breakdown of VM building blocks into services, Fig. 2 maps these abstractions onto the package and directory structure of two Java-based implementations of VMs. This mapping is outlined according to the legend in the figure. This scattered implementation of core abstractions is consistent with an aspect-oriented perspective that, for any given decomposition of a system, there inherently arise multiple fundamental crosscutting concerns.

Clearly, both C- and Java-based approaches show similar characteristics with respect to the challenge of modularising these core VM abstractions. Tab. 1 further categorises the interactions between memory management and scheduling with the rest of the core services that comprise key abstractions within all VMs. Tab. 2 provides a lower-level view of the service interactions with threading in a selected subset of VM implementations.

2.2.1 Service Interaction

Listings 1 and 2 provide concrete code examples of these interactions at the level of individual methods. The method shown in Lst. 1, originating from the OpenVM [22] is from the `Processor` class located in the `ovm.core.execution` package. The location and name of this class imply that it is an integral part of the execution service in the VM. As we delve into the internal workings of this class, we find that other services, including memory management and scheduling, are intersecting at the fine granularity of a method. The `run` method highlighted here evokes the interpreter to kick off the execution service (line 2), but the interpreter itself requires the context of the scheduled thread along with the current processor to be passed to it. The invocation of the interpreter is followed by the registration of that context with the memory manager before any memory is allocated in that thread (line 3).

Lst. 2 is also pulled from a class file in the `ovm.core.execution` package. This class, `Engine`, can likewise be categorised as part of the execution service of the system; but like with `Processor`, the code in *Engine.java* is entangled with code from other core services in the system including but not limited to representation, memory management and

```
1 static public synchronized Code dynamicCompile(S3ByteCode code) {
2   Code oops = code.getMethod().getCode();
3   if (oops != code)
4     Domain d = code.getMethod().getDeclaringType().getDomain();
5   if (((S3Domain) d).sj != null) {
6     Object r = MemoryPolicy.the().enterHeap();
7     try {
8       d.getRewriter().ensureState(code, S3ByteCode.REWRITTEN);
9       return SimpleJITDynamicJITCompiler.dynamicCompile(code);
10    } finally {
11      MemoryPolicy.the().leave(r);
12    }
13  } else {
14    throw Executive.panic("can't␣compile␣" + code.getSelector());
15  }
16 }
```

Listing 2: Interaction between three subsystems in one method.

	Memory Management	**Scheduling**
Representation	object layout, GC info in object headers	lock fields in object headers
OS Interface	heap allocation, paging mechanism	green/native threads
Scheduling	stop-the-world, concurrent GC	N/A
Execution	JIT compiler computing GC maps, stack layout	pre-emptive scheduling
Adaptive Optimisation	object locality	optimisation thread, on-stack replacement

Table 1: Key interactions between memory management and scheduling and the other core VM services.

adaptive optimisation. This method, while dealing with dynamic compilation, requires representation services to find the method (line 2) and its type (line 4). The memory manager is also involved when the execution is forced to enter and exit the heap memory space associated with the memory policy around the JIT dynamic compilation of the code (lines 6, 11).

The Jikes RVM [2, 3] is a rich testbed for research in the area of advanced virtual machine technologies. It has particularly been a hotbed of research in the areas of memory management and adaptive optimisation. Part of the Jikes RVM architectural overview [23] centers on the optimisations of the M:N threading model implemented by inserting yield points at principled points throughout the system and associating required functionality at those points in terms of related services.

The `perform` method shown in Lst. 3 is located in the file *OPT_YieldPoints.java* in the *com/ibm/jikesrvm/opt* directory. This method shows the ways in which yield points may be inserted into method prologues, loop heads and method exits, as long as the code is not *uninterruptable* in support of the multithreading model in the Jikes RVM. The ensuing yield point code sequence is a combination of processor flag checks and scheduler invocations. This threading support is also entangled with memory management (lines 8,16), as all threads must reach a safe point before a collection cycle occurs.

```
1  public final void perform (OPT_IR ir) {
2    if(!ir.method.isInterruptible()) {
3      return;
4    }
5    // (1) Insert prologue yield point unconditionally. As part of prologue/
6    // epilogue insertion we'll remove the yield points in trival methods that
7    // otherwise wouldn't need a stackframe.
8    prependYield(ir.cfg.entry(), YIELDPOINT_PROLOGUE, 0, ir.gc.inlineSequence);
9    // (2) If using epilogue yield points scan basic blocks, looking for returns
10   // or throws
11   if(VM.UseEpilogueYieldPoints) {
12     for(OPT_BasicBlockEnumeration e = ir.getBasicBlocks(); e.hasMoreElements();) {
13       OPT_BasicBlock block = e.next();
14       if (block.hasReturn() || block.hasAthrowInst())
15         prependYield(block, YIELDPOINT_EPILOGUE, INSTRUMENTATION_BCI,
16           ir.gc.inlineSequence);
17     }
18   }
19 }
```

Listing 3: Logic for inserting yield points in JIT-compiled code in the Jikes RVM.

VM	Service	Total Files	Service Files	Interacting Modules
KVM	pre-emptive threading asynchronous support	64 C files	2 C files	47 modules
Kaffe	threading, 6 models	176 C files	25 C files	8 modules
RVM	M:N threading model	664 Java files	25 Java files	76 Java files
HotSpot	threading, 7 models	604 C++ files	21 C++ files	35 C++ files
CSOM (cf. Sec. 4)	threading, green and native	66 C files 4 aspect files	5 C files	22 functions

Table 2: Coarse-grained analysis of threading in various VMs using simple *grep* commands.

3 Disentangled Virtual Machine Architecture

In this section, we will propose an approach for VM architectures. Our approach 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.

This section first presents a high-level overview of the architectural approach that we propose. After that, we will briefly present the structure of VM architecture descriptions using VMADL, the proof-of-concept architecture description language that is used throughout the case studies in the following section.

3.1 Architectural Approach

For illustration purposes, we will use a VM consisting of six subsystems as introduced in Sec. 2: application representation, scheduling, memory management, execution, adaptive optimisation, and operating system interface. Seen at a high level of abstraction, the typical structure of an implementation of such a VM can be depicted as in Fig. 3.

Each of the shapes in Fig. 3 represents the source code of one of the subsystems. The different subsystems are mostly modularised, but parts of their implementations are scattered across the system and tangled within source code pertaining to other subsystems.

Our approach to VM architecture focuses on establishing a clear modularisation for VM subsystems. This is achieved by applying aspect-oriented programming techniques in a pragmatic way. 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. Some fundamental concerns, such as object layout, are *not* modularised, but treated as "base code". This allows more controlled and focused changes which have influences throughout the whole system. We sacrifice some clarity for tangling, but we gain a more agile VM implementation in return. This will be illustrated later on.

Service Modules 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 [13, 14]). Hence, the interface is *bidirectional*. Service modules share this property with *open modules* [14].

An illustration of this implementation principle is shown in Fig. 4a. A service module is a superstructure comprising of several actual implementation modules (e. g., C source files) and may 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, 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 architectural 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.

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 (cf. Sec. 3.2). An exposed point of interest comes with the according context information to be exploited by client services.

Composing 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 realised 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 points of interest exposed from service modules, which are called *service-exposed join points*.

Fig. 4b shows, at the same level of abstraction as Fig. 3, 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.

Thus, AOP techniques are applied in this model at two degrees of abstraction, namely by introducing different join point models both inside service modules and at the level of service module interaction. 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 through advising. The two join point models found in the architecture approach are separate from each other. Service-exposed join points can be assigned meaningful names, constituting *domain-specific* join point models that improve declarativeness at the architectural level.

Discussion Among the advantages of this approach, it is clearly worth mentioning that VM subsystem interactions can be expressed very cleanly. Service module interfaces concisely describe, on the one hand, the set of service requests client modules may send to a module, and, on the other, the set of join points exposed from within the module to which clients may attach. Actually, through the use of service-exposed join points, a user-level or, respectively, domain-specific join point model is created that allows for declaratively describing service interactions.

Clean interfaces that do not impose assumptions regarding module internals on clients improve modularity in that they help decoupling the various subsystems from each other. Given the high and intricate complexity of subsystem interactions in VM implementations, this is especially welcome. Moreover, increased decoupling is likely to increase reusability of single modules, and even reconfigurability of entire VM architectures or parts thereof. Sec. 6 gives information on how we plan to address this.

Of course, service module providers are responsible for exposing the right join points that clients may require to attach to. Join point exposure also requires detailed knowledge about service module internals, which may not always be available, e. g., when a new join point needs to be exposed from a third-party module.

One may also ask whether the extensive use of join points, pointcuts and advice in implementing service module interactions degrades performance, which is critical in the domain of VM implementations. While this concern is understandable, it is clearly the responsibility of the AOP implementation to provide the required performance characteristics. It has been shown that the employment of AOP techniques does not have to have a negative impact on performance at all [24]. Also, our pragmatic adoption of AOP ideas explicitly deals with efficiency concerns in the core of the VM implementation.

3.2 Applied Disentangling

In this section, we give an overview of *VMADL* (*virtual machine architecture description language*). It is an aspect-oriented ADL that introduces some constructs for describing VM architectures in terms of service modules and their interactions. VMADL constructs are imposed on the actual language used for a particular VM implementation, and to an aspect language that can be used in conjunction with said implementation language. Thereby, VMADL acts as a front-end to both the implementation and the aspect language; it is not conceptually restricted to, e. g., Java- or C-like languages.

The examples used throughout this paper stem from a VM implementation written in C (cf. Sec. 4). Aspicere2 [25, 26] was used as the aspect language. Its pointcut and advice syntax resemble those of AspectJ [6, 27], hence the examples should be easily accessible to readers familiar with that language.

Service Module Structure A service module is constituted by two main parts: all implementation modules (such as classes in object-oriented programming languages) on the one hand, and a service module declaration on the other.

The former are not special; they constitute the functionality of the respective service or VM subsystem. The actually interesting part is the service module declaration, which is responsible for determining the set of classes that make up the module, for declaring service-exposed join points, and for declaring interaction with other service modules.

The rough structure of a VMADL service module declaration is shown in Lst. 4. Depending on the VM implementation language at hand, the service module declaration may start with a number of **`import`** statements, **`#include`** directives or the like that serve the same purpose as in the corresponding implementation languages.

A service may have "global" state, i. e., state that is global with respect to the service and should hence be accessible from all implementation modules that constitute it. Such global state is declared like it would be in a plain implementation module. The **`expose`** section specifies a list of service-exposed join points which can be advised by other services. What follows is a list of interactions between this service and the join points exposed by other services. Finally, the **`startup`** section specifies which operations must be performed in order to get this service running. It may start with a method which

```
// may be preceded by import statements, #include directives, etc.
service MyService {
  // declarations of any service-wide "global" state

  expose {
    // list of service-exposed join points, complete with name and context
    // parameters
  }

  // list of interaction specifications denoting to which service-exposed join
  // points this service reacts

  startup {
    // optional entry point that has to be invoked to get this service running
    boot void initialize_me();

    expose {
      // list of startup-time service-exposed join points
    }

    // list of startup-time interaction specifications
  }
}
```

Listing 4: Structure of a VMADL service module declaration.

should automatically be invoked and it may expose and advise join points which will only be active during startup-time. If a join point should be exposed both during startup-time and normal execution-time, the join point declaration is preceded by the **`always`** modifier.

Example To illustrate how interaction specifications by means of bidirectional interfaces may look in practice, Lst. 5 shows fragments of two service modules `Execution` and `GC`. The example is based on the code given in Lst. 2 on p. 7 and shows how the crosscutting relationships found in that code can be dealt with.

The example shows how both startup and run-time join point exposure are employed to coordinate interaction between the execution and garbage collection services. During startup, the execution service exposes a join point denoting when the interpreter boots (lines 4–6); when this join point is signalled, some context information is passed along that can be used in attached advice (lines 19–22).

Later on, the execution service exposes a join point signalling that method compilation is taking place (lines 10–12). Method compilation effectively starts with the invocation of `Domain.getRewriter()` and ends after the call to the JIT compiler's `dynamicCompile()` method completes. The **`span`** construct covers this kind of join points that actually denote a whole range of instructions. As the garbage collection service shows (lines 24–29), advice can be attached to such join points in the normal way.

Implementation VMADL's realisation features a preprocessor that transforms VMADL specifications into constructs—aspects, pointcuts, advice—of the target language. In the

```
1  service Execution {
2    startup {
3      expose {
4        interpreterStartup(Context ctx):
5          call(void Interpreter.run(proc, *)) where
6            ctx = Interpreter.getContext(proc);
7      }
8    }
9    expose {
10     compilation():
11       span(call(* Domain.getRewriter()),
12            call(* SimpleJITDynamicJITCompiler.dynamicCompile(..)));
13   }
14   // ...
15 }
16
17 service GC {
18   startup {
19     after(Context ctx):
20       Execution.interpreterStartup(ctx) {
21         MemoryManager.the().observeContextSwitch(ctx);
22       }
23   }
24   Code around():
25     Execution.compilation() {
26       Object r = MemoryPolicy.the().enterHeap();
27       try { proceed(); }
28       finally { MemoryPolicy.the().leave(r); }
29     }
30   // ...
31 }
```

Listing 5: A possible application of service module abstraction to the OVM.

case study in Sec. 4, C is the target language used for the VM implementation, and Aspicere2 [25, 26] is used as the AOP language. As VMADL tries to be aspect-language agnostic, there can be various language back-ends.

4 Case Study

In this section, we present a case study in which we have applied the principles described in this paper. The case study revolves around CSOM, a VM used in teaching, and deals with the integration of garbage collection and multithreading. We first briefly present CSOM and its architecture before we move on to describing and evaluating how the various extensions have been achieved.

4.1 CSOM

CSOM is a VM for a Smalltalk dialect which is used for teaching purposes. Its precursor, SOM (Simple Object Machine) was developed at the University of Århus, and was implemented in Java. CSOM is the port of SOM to C that has been done at the Hasso-Plattner-Institut, where it is used in courses on VMs.

The architecture of CSOM is deliberately simple to allow for its use in teaching. Fig. 5(a) shows the high-level service decomposition of CSOM, whereas Fig. 5(b) zooms in on the source code dependencies between the various services. CSOM consists of four services: a simple bytecode interpreter, an object model for representing Smalltalk entities, a memory management service and the universe. The interpreter implements the interpretation of the Smalltalk bytecodes. CSOM's object model provides a number of operations such as accessing the fields of an object and sending messages, and it implements a number of native classes, such as numbers, tables and activation records. The memory service is rather trivial, as it simply relies on C's `malloc` and performs no automatic memory management. The universe maintains global Smalltalk objects (primarily all the classes in the system).

Each of these services comprises one or more implementation and header files. The standard implementation is completed with a Smalltalk parser and compiler, and a standard library of roughly two dozen classes. Neither automatic memory management nor JIT compilation are supported. The CSOM source code consists of 76 C files (37 source and 39 header files) accounting for 4753 SLOC[1]. Although CSOM is a rather simple VM, we already encountered crosscutting during our experiments.

In our case study, CSOM has been extended with four extra services, two for garbage collection and two for multithreading. We have opted for these two kinds of services because garbage collection is a very crucial concern, whereas one can in principle do without multithreading. We did not combine multiple garbage collection and/or multithreading services at the same time, as we did not focus on resolving interactions.

[1]The number of non-blank, non-comment lines of C code according to `http://www.dwheeler.com/sloccount/`.

	number of new...			number of modified...				#added
	.c	.h	lines	.c	.h	lines	loci	lines
reference counting	0	0	0	15	3	22	37	170
mark-sweep	0	0	0	7	2	39	25	332
native	3	3	268	5	5	3	13	333
green	1	1	148	5	3	2	9	78

Table 3: Data for CSOM with each of the four extensions.

Instead, we have made eight different versions of CSOM: four in which one garbage collection or multithreading service has been implemented *without* VMADL, and four in which we have used a VMADL description to integrate a garbage collection or multithreading service. The versions of CSOM that were built using AOP techniques show no differences in observable behaviour from the tangled implementations. The next sections present VMADL code fragments for the services, followed by a discussion of interesting issues related to them.

Performance was, given the proof-of-concept state of the implementation, only cursorily evaluated. The test suites and benchmarks that are included with CSOM were run to get an impression of the performance impact due to using AOP techniques. No notable overheads were observed. We expect this to be true as well for a future full implementation, which will be evaluated thoroughly.

4.2 VMADL Applied to CSOM

This section presents the VMADL implementation of the four garbage collection and multithreading services we have integrated with CSOM one at a time. The interactions between the basic CSOM services on the other hand are not of such a complexity that they need the advanced constructs of VMADL. It is mainly the introduction of proper memory management and multithreading services that cause the high degree of crosscutting. Therefore, we present the VMADL implementations of the object model, interpreter and universe services after introducing memory management and multithreading, so that we learn which join points should be exposed along the way. The code for these services is shown in Lst. 8.

4.2.1 Garbage Collection in CSOM

Two garbage collection implementations have been realised, namely a *reference counting* and a *mark-sweep* collector [28]. Both implementations are very interesting for studying crosscutting concerns in VM implementations, not only because they are, as memory managers, naturally closely interwoven with the rest of the system, but also because they follow very different approaches.

On the one hand, a reference counting collector requires the VM to update the reference counts of VM objects whenever pointer assignments are made in the executed program. As soon as the interpreter executes an assignment of the form `x := y`, the

reference count of the object referenced by `y` must be incremented because `x` now references the same object. Conversely, the count of the old object referenced by `x` must be decremented. Garbage collection is done on the fly when reference counts reach zero.

On the other hand, a mark-sweep collector does not observe all reference updates, but instead checks, upon allocation requests, whether memory is exhausted. If so, garbage collection is performed before allocation is done. Collection is carried out in two phases, during which the application execution is paused. In the first phase, all live objects are marked starting from a root set comprised of global constants (e. g., loaded classes) and the current stack frame. In the second phase, the heap is traversed, and all unmarked objects, i. e., those that are no longer live, are de-allocated.

Reference counting and mark-sweep collectors exhibit significant differences in the degree of crosscutting they introduce in a VM implementation. While a mark-sweep collector essentially only attaches to allocation requests and operations that affect the root set, a reference counter interacts with all parts of the VM that manipulate object references. This is apparent when comparing an initial version of CSOM without automatic garbage collection with a tangled, non-AOP version with either reference counting or mark-sweep. Tab. 3 shows some metrics for this, and Fig. 6(a) shows the architectural changes for the tangled reference counting implementation. Neither reference counting nor mark-sweep have introduced new files, although at least 170 SLOC have been added to existing files. The last five columns of Tab. 3 very clearly show the different crosscutting natures of reference counting and mark-sweep. For the former, CSOM has been modified in 37 places spread across 15 *.c* and 3 *.h* files, and 22 existing lines have been changed. Fig. 6(a) shows that each service has been altered. Mark-sweep, on the other hand, has led to changes in only 25 places in 7 source and 2 header files, all of which belong to the memory and object model services. Still, 39 original lines were modified.

For reference counting, the biggest changes involve the addition of an extra reference count field to the object layout and of the logic to update this counter. The latter consists of scattered calls to the new counter increment and decrement functions in the supporting data structures (lists, abstract syntax tree nodes in the compiler, etc.) as well as in the two functions which handle assignment to object fields and array elements. Extra care is taken to handle stack push and pop operations. In the case of mark-sweep, changes are far less wide-spread, as only the object layout has been extended with one extra bit, the (de-)allocation functions have been rewritten and logic has been added for determining the root set.

The code in Lst. 6 shows part of the service implementation for the reference counting garbage collector. The assignment of a new value to a certain field in an object is advised in order to properly deal with the reference counts: the new value's reference count must be incremented; the old value's, decremented—possibly triggering de-allocation. The pointcut expression uses a join point exposed from the `ObjectModel` service (cf. Lst. 8).

Fig. 6(b) displays the influence of the VMADL specification for reference counting on the CSOM architecture. Instead of requiring changes to each service, only the object model (extra field) and memory management have been changed. The other changes of Fig. 6(a) have been replaced by advice on the interpreter, universe and object model services. Hence, the VMADL specification has enabled a more modular extension of

```
1 service RefCountGC {
2   ...
3   around(VMObject object, int index, VMObject newValue):
4     ObjectModel.object_assignment(object, index, newValue)
5   {
6     // store the old value temporarily
7     VMObject oldValue = object_get_field(object, index);
8     proceed();
9     increment_reference_count(newValue);
10    decrement_reference_count(oldValue);
11  }
12
13  after(VMObject object): Universe.new_global_object(object) {
14    // mark the object as "global", i.e., immortal
15    gc_set_global_object(object);
16  }
17
18 }
```

Listing 6: Reference counting implementation in VMADL.

CSOM in the case of reference counting. For the mark-sweep service, improvements are not that significant, as this service only requires changes to the object model and memory management services.

4.2.2 Multithreading in CSOM

As for garbage collection, two approaches have been realised for multithreading as well: *native* threading using *POSIX threads* (*pthreads*) [29, 30], and *green* threading, i. e., a realisation of threading that resides entirely in the VM implementation itself.

Again, both approaches show different crosscutting characteristics. The implementation of native threads—basically allowing for multiple instances of the interpreter running in parallel—calls for all accesses to interpreter state being transformed into accesses to thread-local storage. This affects the entire interpreter implementation.

Conversely, green threading only requires a few extensions to the interpreter to allow for context switches at well-defined points. The green threading implementation in CSOM performs context switches when the interpreter executes return instructions.

Again, we compare a tangled implementation of both multithreading implementations with a VMADL implementation (cf. Tab. 3). Multithreading clearly shows to be less invasive than garbage collection, as changes are localised in some new files and in a limited number of places in the original code. Native threading requires slightly more changes and almost four times as much extra code. Fig. 7(a) shows the service architecture for native threading. The threading logic is explicitly invoked from within the interpreter and the universe. The latter two services are also the only ones which are altered by the tangled implementation of green threading.

Unlike the garbage collection implementations, the two threading versions both introduce new source files (scheduler, signals, etc.) or—in case of native threading—dependencies on external libraries. Native threading requires safe access to the current

```
1  service GreenThreading {
2    startup {
3      after(): Universe.vm_setup() {
4        // load and instantiate scheduler classes
5        ...
6      }
7      after(String core_file): Universe.load_primitives(core_file) {
8        // load the primitive methods on the scheduler classes
9        ...
10     }
11   }
12
13   after(): Interpreter.return_bytecode() {
14     reschedule();
15   }
16   ...
17 }
```

Listing 7: Green threading implementation in VMADL.

frame and thread via explicit getters and setters to synchronise access between threads, while the global variables themselves have to be made thread-local. The core of this threading approach is the code which handles spawning a new thread for the execution of a code block. Finally, care has to be taken to initialise and clean up threads. Green threading is much simpler, as one does not rely on real operating system support for it. Here, the most basic action is the calling of the scheduler after execution of a return bytecode to possibly yield control to another thread. The other modifications merely add extra state and threading functionality.

The code in Lst. 7 shows part of the green threading service implementation, beginning with a `startup` section containing two pieces of advice. Both pointcuts used are exposed from the `Universe` service. The advice following the **`startup`** yields control to the scheduler after every return bytecode executed by the interpreter.

The code in Lst. 8 shows parts of the implementations of the `Universe`, `ObjectModel` and `Interpreter` services, more precisely the definitions of the exposed join points that are used by the reference counting and green threading services. The pointcut definitions may refer to private members of the service implementation since other services only see it as one "primitive" join point of the user-level join point model.

Fig. 7(b) shows the service architecture of CSOM with the VMADL implementation of native threading. No default service of CSOM has been altered. Instead, the new native threading service is integrated with CSOM via advice (two dotted edges). The green threading implementation achieves the same effect. Hence, for the multithreading services VMADL enables us to extend CSOM in a modular way without requiring changes to the base services.

4.3 Discussion

As observed for the garbage collection and multithreading services, domain-specific abstractions and bidirectional interfaces make composing a VM implementation easier.

```
1 service Universe {
2   startup {
3     boot void universe_initialize();
4     expose {
5       vm_setup(): execution(initialize_globals());
6       load_primitives(char* coreFile):
7         call(dlopen(corefile, *));
8       ...
9     }
10    ...
11  }
12  expose {
13    new_global_object(VMObject object):
14        execution(symbol_table_insert(object))
15     || execution(set_global(*, object));
16    ...
17  }
18  ...
19 }
20
21 service ObjectModel {
22   ...
23   expose {
24     object_assignment(VMObject object, int index, VMObject newValue):
25       execution(object_set_field(object, index, newValue));
26     ...
27   }
28 }
29
30 service Interpreter {
31   ...
32   expose {
33     return_bytecode():
34       (call(do_return_local()) || call(do_return_nonlocal()))
35       && withincode(interpreter_start);
36     ...
37   }
38   ...
39 }
```

Listing 8: VMADL implementation (parts) of the `Universe`, `ObjectModel` and `Interpreter` services.

First, every service offers a clear and coherent functionality. Fundamental concerns which have not been modularised are easily identifiable, too. Second, each service explicitly states which other services it uses, either for start-up or at run-time, and what functionality it offers. Understanding is favoured by the use of concepts of the VM domain in the service specifications.

Multithreading has been very straightforward to implement, as it did not touch on core data structures or concepts of the VM. Garbage collection, on the other hand, has raised two important issues, which we now discuss.

Structural Crosscutting in Object Layout Both garbage collection services require changes to the object layout. This could be handled by applying inter-type declarations (ITD) to add some fields to the respective C `struct`. However, as object layout is crucial to a VM, fine-grained control over the placement of new fields is indispensable. Mark-sweep only requires one extra bit, so sloppy programming could potentially add one extra word to every object. Dealing with this using ITD requires a very powerful, complex pointcut language capable of dealing with conflicting introductions of state. If another service also requires an extra bit, there could be a fight to get the most favourable bit location.

The second approach would be a data structure like a hashmap or bitmap inside the new service itself, which is ideal for conflict resolution as the new state is private to the service. However, memory is wasted this way, and lookup mechanisms cause overhead. In addition, garbage collection should know about the full object layout in order to clean up objects, hence extending the layout in a localised way does not work.

As mentioned before, object layout is crucial to a VM. To effectively apply AOP, the requirement of modularising *everything* inside the VM should be relaxed. Instead, this case seems to point out that it is advantageous for some very fundamental concerns in a VM implementation to leave things tangled. Concretely this means that for example in the VMADL version of the reference counting extension, the interaction with the interpreter is expressed using AOP, but the field for storing the reference count is still manually added to the object model.

Finding Roots Finding roots for mark-sweep garbage collection requires knowledge of all global and local variables in existence when the mark phase starts. In the extreme, this could mean that every service needs to expose all its global variables together with every combination of method execution and a local variable. In practice, this boils down to classic, unrestricted AOP, as if no VMADL interface mechanism was in use. We therefore did not express this interaction in VMADL, as it would defeat the point.

A second possibility of dealing with this is via the notion of critical sections. VM code would be designed from scratch in such a way that, when allocation is requested, previously allocated local variables will not be used anymore, as they are unsafe to use. This requires *a priori* knowledge about possible garbage collection schemes which will ever be used. Also, not all schemes make use of this, e. g., reference counting.

Summary The preceding paragraphs show that, even with domain-specific abstractions in place, important issues remain to be dealt with. We would like to stress that a dogmatic adoption of AOP techniques is, in our opinion, not beneficial. However, the mechanisms we propose significantly improve modular abstraction and communication between modules, as observed in Fig. 6(b) and 7(b). By applying VMADL, we have been able to express several crosscutting concerns in a more modular way, reducing scattering and tangling in our VM implementation.

5 Related Work

The work presented here greatly benefits from both lessons learned and recent advances from the systems and AOSD communities, respectively.

The systems software kernel MACH [31] exhibited a modular infrastructure key to its improved scalability and extensibility relative to its monolithic counterparts. By better modularising key abstractions such as tasks, threads, and memory objects, a spectrum of scheduling and virtual memory policies could be realised from these core mechanisms. Further, microkernel architectures, such as Exokernel [32] and Nemesis [33] echoed the need for identifying the right abstractions and exposing the right interfaces within the kernel in order to support OS services in user space. We not only share these same motivations, but also the sense that current VM infrastructures suffer from their monolithic nature. In operating systems, this alternative architecture revealed the precise ways in which simple vertical-layering fails to capture service interaction.

Operating systems such as SPIN [34] set the stage to consider issues such as granularity and composition of extensions in OS code. Instead of moving OS services to user space, these systems effectively introduced user code to kernel space. Similarly, work aimed at extending VMs [35] considers ways in which lower-level interfaces can be safely accessed by user applications. A common theme in all of this work is the need to have the right abstractions and interfaces exposed to support the composition of services.

The systems community is also exploring language-based solutions to the problem of making interactions explicit according to domain specific needs. NesC [36] is an extension to C designed to capture key abstractions in TinyOS [37]. Components specify behaviour according to bi-directional interfaces that are explicitly provided or used by a given component. A NesC interface declaratively identifies an interaction between components based on the ability to register interest in an event and to receive a callback when an event happens. In keeping with the approach used in NesC, we would like to separate construction and composition. Not only does this approach promote a means of more easily assembling a system from services, but it enables capturing and making explicit complex interactions between services.

More general component models from this community include the THINK framework for building OS kernels [38]. This framework is an implementation of the Fractal component model, which enables system designers to specify which operations from other components a given component requires when a given situation arises. It is the explicit representation of these kinds of interactions that we share in common with the

motivation of the work on THINK.

Previous work on architecture languages includes Darwin [39] and ArchJava [40]. Darwin is a declarative binding language used to define hierarchic compositions of interconnected components involved in distributed applications. Central abstractions managed by Darwin are services and components, where services are the means by which components interact. ArchJava is an extension to Java that uses architectural specifications similar to those in Darwin. ArchJava explicitly couples software architecture with implementation by leveraging a type system to ensure that code conforms to architectural constraints. In general terms, we share with these systems the motivation to make services and architectural specifications an explicit feature of system compostion.

Service modules' bidirectional interfaces as met in VMADL are conceptually very close to both crosscutting interfaces (XPIs) [13] and open modules [14]. Both support the exposition of pointcuts; open modules moreover exhibit an API for inbound requests. Matches of an open module's exposed pointcuts, or calls to one of its API methods, can be advised, but not its internal implementation. Moreover, both XPIs and open modules essentially express design rules that improve decoupling in a system.

The main difference between bidirectional interfaces on the one hand, and XPIs and open modules on the other lies in the domain-specific abstraction of run-time stages that service modules support: certain join points are exposed, and certain service module interactions take effect at startup time *only*, which is declaratively specified. Bidirectional interfaces are also different from XPIs in that the latter merely serve to expose pointcut definitions [13], while the former support the exposure of join points, the specification of interactions with other modules, and API definitions. In that regard, bidirectional interfaces are actually a variant of open modules that has been augmented with domain-specific declarative means.

DAOP-ADL [41] is an aspect-oriented architecture description language. It is geared towards component-based applications. While it allows for describing application architectures, the granularity of its join point model is that of messages being sent between components. DAOP-ADL does not allow for declaring join points to be signalled from within a component; all interaction is solely described in terms of the interfaces of components.

Research on VM architectures with an emphasis on improving modularity is rare. Three projects that deserve attention are JavaInJava [42], JnJVM [43], and ORP [44]. The JavaInJava project was conducted as an internal research project at Sun, with the goal of developing a cleanly modularised Java VM in the Java programming language. JavaInJava is extremely simple, featuring a plain bytecode interpreter and entirely relying on the host JVM for memory management—it is not surprising that the achieved architecture is reported to be well-modularised and understandable [42].

JnJVM and ORP both rely on strict and clean interfaces in their architectures, and they do so for similar reasons: ORP is a research platform designed for flexibility, JnJVM is intended to be tailorable by VM plugins that applications bring along for specific optimisations. In both cases, however, the interfaces that are used are unidirectional APIs; the degree of module interaction support is naturally lower than that brought about by VMADL.

Finally, the way VM implementations are organised using bidirectional interfaces—with modules exposing join points other modules can advising—resembles the organisation of publish/subscribe system architectures. We argue that this resemblance is at a very high level of abstraction only. Publish/subscribe systems usually have some form of first-class representation of events that is available at run-time, and that requires an infrastructure to handle and manage events. Since a core concern of VM implementations is performance, having a sophisticated event handling infrastructure is not an option. Performance is achieved through weaving techniques and implicit handling of join points.

6 Summary

We have presented an analysis of crosscutting concerns in VM implementations and given examples for them. Motivated by our observation that VM implementations consist of many intricately combined crosscutting concerns, we have proposed an approach to organising VM architectures based on *service modules*, i. e., modules that each implement one well-defined service that the VM provides to applications running on it.

Interactions among service modules are organised using *service-exposed join points*—points of interest signalled from within a service module. They are defined using aspect-oriented means. The coordination of module interaction is also realised via AOP techniques, allowing for the expression of VM architectures with explicit dependencies. We have given several examples supporting our approach, and demonstrated its feasibility in a case study featuring a simple yet functional VM implementation. The systems in the study were realised using a preliminary proof-of-concept version of VMADL, a language derived from the presented approach.

In our ongoing work, we are concerned with the investigation of how the application of the proposed architectural concepts can affect future VM development. An important insight that we have gained is that refactoring existing VM code to VMADL-based implementations is tedious, if not infeasible. As a matter of fact, VM implementations like HotSpot or Strongtalk have not been designed and implemented with a clean and modular architecture as their main objective. A VM implementation that faithfully adopts our proposed architecture will most likely have to be developed from scratch. Therefore, it is a core interest of our ongoing and future work to gain a better understanding of the requirements on service module interfaces.

We expect VM architectures based on the principles discussed herein to exhibit interesting properties. In particular, we will investigate in how far the increased decoupling and disentangling gives rise to regarding VM architectures from a product-line perspective, including the expression of variabilities and tailorability of concrete VM implementations in a product line. Part of our research will consequently concentrate on establishing common interfaces for VM services.

It is also interesting to assess the architectural soundness of VMADL specifications, exploiting the interaction descriptions that can be directly derived from such specifications. To that end, it will have to be investigated how single service modules can be

validated, and how such results can be combined to validate the entire architecture.

Acknowledgements

The authors are grateful for the valuable contributions to and comments on this paper that were made by Jan Vitek, Wolfgang de Meuter, Theo D'Hondt, and Pascal Costanza.

Bram Adams is supported by a BOF grant from Ghent University. Stijn Timbermont is supported by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

Martin Beck, Juri Engel, Gregor Gabrysiak, Jan Klimke, Thomas Kowark, Martin Sprengel, and Martin Zahn have worked on the original (tangled) implementations of the four extensions to CSOM during a virtual machines course taught at the Hasso-Plattner-Institut in summer 2006. We thank them for their contributions. We also thank Tobias Pape for his ongoing work on CSOM.

References

[1] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan-Kaufmann, San Francisco, 2005.

[2] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[3] `http://jikesrvm.sourceforge.net/`, accessed March 2008.

[4] `http://java.sun.com/javase/technologies/hotspot/`, accessed March 2008.

[5] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM '04: Proceedings of the 4th International Symposium on Memory Management*, pages 49–60, New York, NY, USA, 2004. ACM.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–353, London, UK, 2001. Springer-Verlag.

[7] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 60–69, New York, NY, USA, 2003. ACM.

[8] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, volume 2487 of *LNCS*, pages 173–188, London, UK, 2002. Springer-Verlag.

[9] C. Allan et al. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 345–364, New York, NY, USA, 2005. ACM.

[10] C. Herzeel, K. Gybels, P. Costanza, C. De Roover, and T. D'Hondt. Forward chaining in HALO: an implementation strategy for history-based logic pointcuts. In *ICDL '07: Proceedings of the 2007 International Conference on Dynamic Languages*, pages 157–182, New York, NY, USA, 2007. ACM.

[11] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*, pages 214–240. Springer, 2005.

[12] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, New York, NY, USA, 2004. ACM.

[13] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[14] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*, pages 144–168. Springer, 2005.

[15] `http://www.kaffe.org/`, accessed March 2008.

[16] `http://harmony.apache.org/`, accessed March 2008.

[17] `http://tinyvm.sourceforge.net/`, accessed March 2008.

[18] `http://java.sun.com/products/cldc/`, accessed March 2008.

[19] E. Gagnon and L. Hendren. SableVM: a research framework for the efficient execution of Java bytecode. In *JVM'01: Proceedings of the JavaTM Virtual Machine Research and Technology Symposium*, pages 27–40, Berkeley, CA, USA, 2001. USENIX Association.

[20] `http://sablevm.org/`, accessed March 2008.

[21] `http://lejos.sourceforge.net/`, accessed March 2008.

[22] `http://www.cs.purdue.edu/homes/jv/soft/ovm/index.html`, accessed March 2008.

[23] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *VM'04: Proceedings of the 3rd Virtual Machine Research and*

Technology Symposium, pages 57–72, Berkeley, CA, USA, 2004. USENIX Association.

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

[25] `http://users.ugent.be/~badams/aspicere2/`, accessed March 2008.

[26] B. Adams and K. De Schutter. An aspect for idiom-based exception handling: (using local continuation join points, join point properties, annotations and type parameters). In *SPLAT '07: Proceedings of the 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies*, pages 1–8, New York, NY, USA, 2007. ACM.

[27] `http://www.eclipse.org/aspectj/`, accessed March 2008.

[28] R. Jones and R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management.* Wiley, Chichester, 1996.

[29] `http://www.unix.org/single_unix_specification/`, accessed March 2008.

[30] B. Lewis and D. J. Berg. *Threads Primer. A Guide to Multithreaded Programming.* Prentice Hall, Mountain View, 1996.

[31] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

[32] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pages 251–266, New York, NY, USA, 1995. ACM.

[33] D. Reed and R. Fairbairns. The Nemesis kernel overview. `http://citeseer.ist.psu.edu/reed97nemesis.html`, 1997.

[34] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility safety and performance in the SPIN operating system. In *SOSP '95: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*, pages 267–283, New York, NY, USA, 1995. ACM.

[35] T. L. Harris. *Extensible Virtual Machines.* PhD thesis, Computer Laboratory, University of Cambridge, UK, 2001.

[36] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The *nesC* language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 1–11, New York, NY, USA, May 2003. ACM.

[37] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in TinyOS. In *NSDI'04: Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 1–14, Berkeley, CA, USA, 2004. USENIX Association.

[38] J. Fassino, J. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *ATEC '02: Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, pages 73–86, Berkeley, CA, USA, 2002. USENIX Association.

[39] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In W. Schafer and P. Botella, editors, *Proceedings of the 5th European Software Engineering Conference (ESEC 95)*, volume 989 of *LNCS*, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

[40] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[41] M. Pinto, L. Fuentes, and J. M. Troya. DAOP-ADL: an architecture description language for dynamic component and aspect-based development. In *GPCE '03: Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, volume 2830, pages 118–137. Springer, 2003.

[42] A. Taivalsaari. Implementing a Java(TM) Virtual Machine in the Java Programming Language. Technical Report SMLI TR-98-64, Sun Microsystems, M/S MTV29-01, 901 San Antonio Road, Palo Alto, CA 94303-4900, 1998.

[43] G. Thomas, F. Ogel, A. Galland, B. Folliot, and I. Piumarta. Building a flexible Java runtime upon a flexible compiler. *International Journal of Computers and Applications*, 27(1):27–34, 2005.

[44] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The Open Runtime Platform: A flexible high-performance managed runtime environment. *Concurrency and Computation: Practice and Experience*, 17(5–6):617–637, 2005.

Figures

List of Figures

1 Directory structure of C-based VMs. 29
2 Directory structure of Java-based VMs highlighting core services. The legend provides a list of core services, which are mapped to the implementation directory location(s). 29
3 Tangled VM architecture. 30
4 (a) Exposing "points of interest" from service modules using pointcuts. (b) Disentangled VM architecture achieved by coordinating points of interest exposed from service modules. 30
5 High-level service architecture of the CSOM VM: (a) in terms of services and (b) in terms of implementation and header files. CSOM consists of an **int**erpreter, **obj**ect model, **mem**ory management and **univ**erse. 30
6 High-level service architecture of the CSOM VM with reference counting: (a) tangled implementation and (b) VMADL implementation. CSOM consists of an **int**erpreter, **obj**ect model, **mem**ory management and **univ**erse. Grey nodes correspond to changed service implementations compared to the CSOM architecture in Fig. 5(a), dashed edges denote new dependencies, and dotted edges represent advice relations. 31
7 High-level service architecture of the CSOM VM with native threading: (a) tangled implementation and (c) VMADL implementation. CSOM consists of an **int**erpreter, **obj**ect model, **mem**ory management and **univ**erse. Grey nodes correspond to changed service implementations compared to the CSOM architecture in Fig. 5(a), dashed edges denote new dependencies, and dotted edges represent advice relations. 31

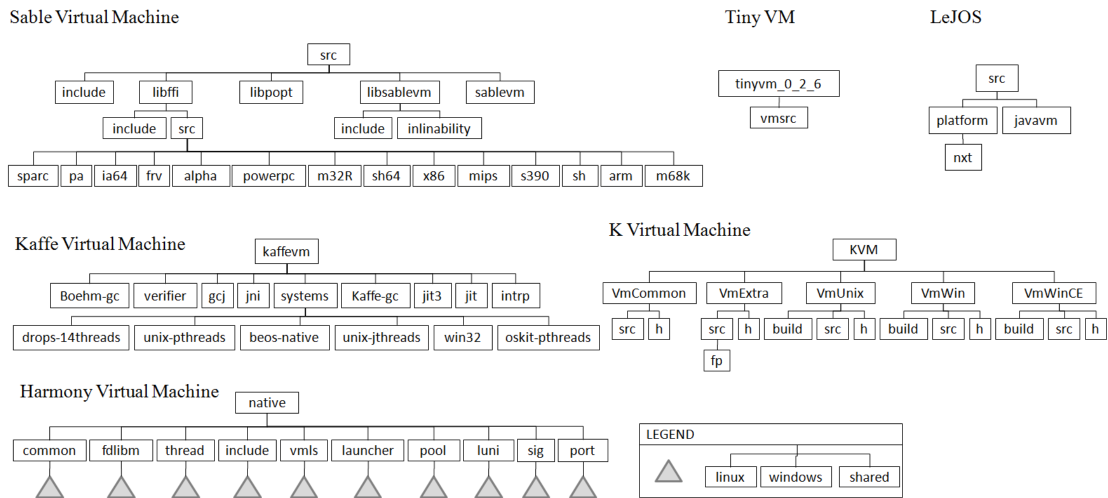

Figure 1: Directory structure of C-based VMs.

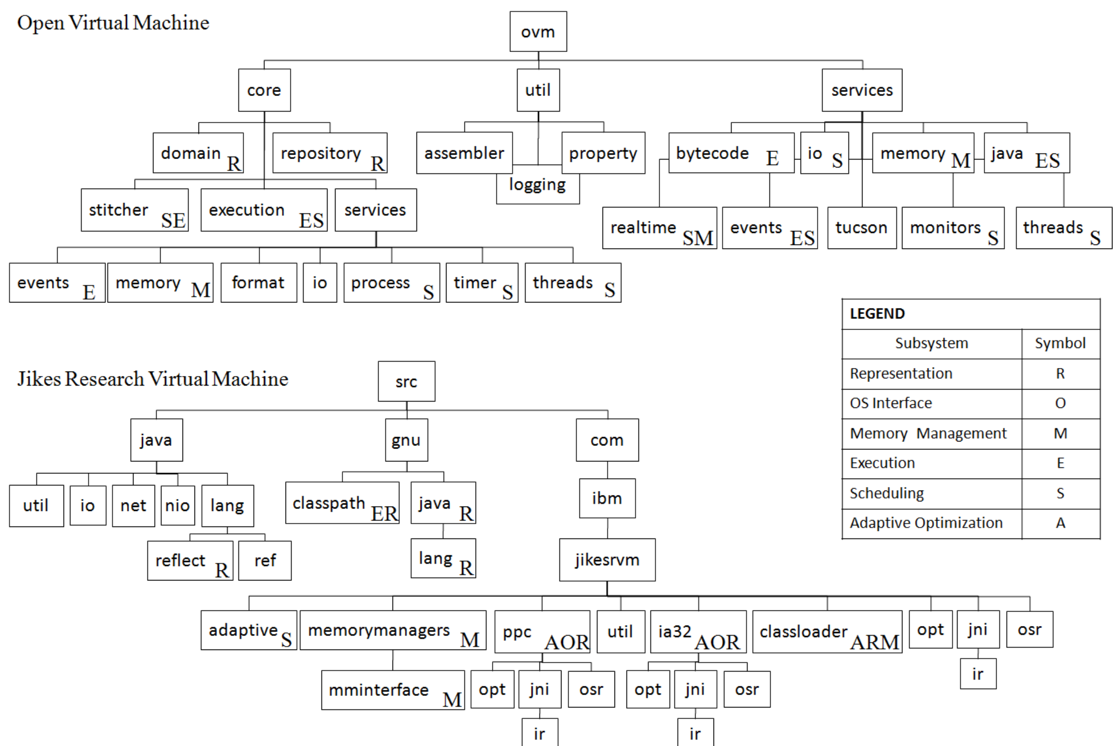

Figure 2: Directory structure of Java-based VMs highlighting core services. The legend provides a list of core services, which are mapped to the implementation directory location(s).

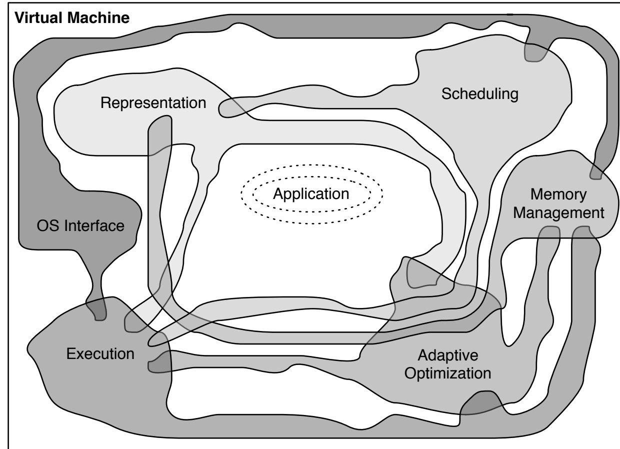

Figure 3: Tangled VM architecture.

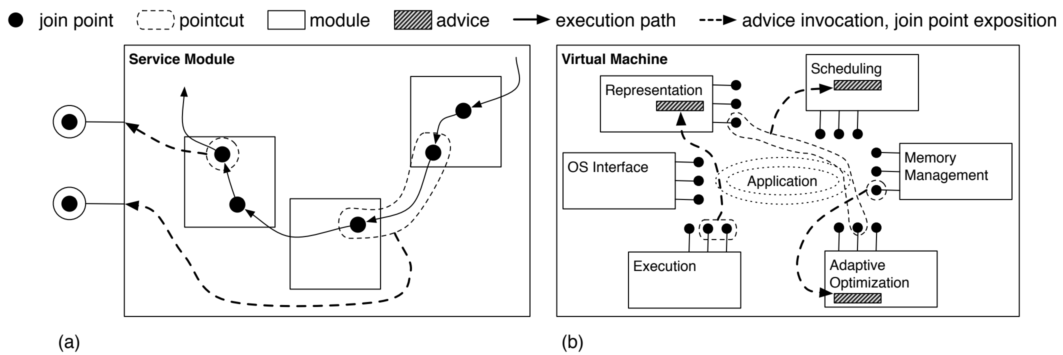

Figure 4: (a) Exposing "points of interest" from service modules using pointcuts. (b) Disentangled VM architecture achieved by coordinating points of interest exposed from service modules.

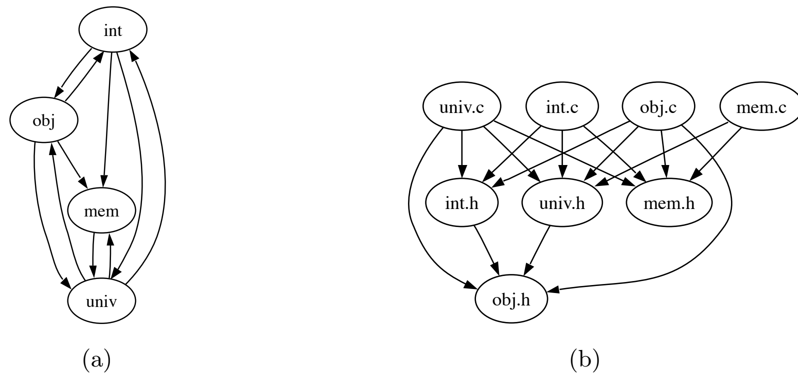

Figure 5: High-level service architecture of the CSOM VM: (a) in terms of services and (b) in terms of implementation and header files. CSOM consists of an **int**erpreter, **obj**ect model, **mem**ory management and **univ**erse.

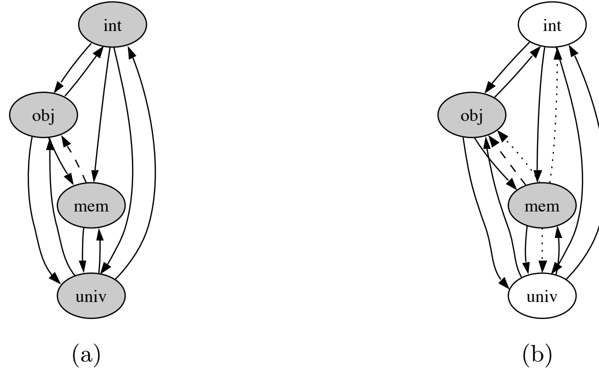

Figure 6: High-level service architecture of the CSOM VM with reference counting: (a) tangled implementation and (b) VMADL implementation. CSOM consists of an **int**erpreter, **obj**ect model, **mem**ory management and **univ**erse. Grey nodes correspond to changed service implementations compared to the CSOM architecture in Fig. 5(a), dashed edges denote new dependencies, and dotted edges represent advice relations.

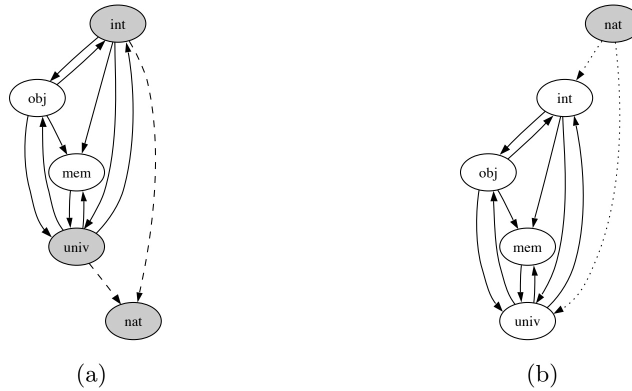

Figure 7: High-level service architecture of the CSOM VM with native threading: (a) tangled implementation and (c) VMADL implementation. CSOM consists of an **int**erpreter, **obj**ect model, **mem**ory management and **univ**erse. Grey nodes correspond to changed service implementations compared to the CSOM architecture in Fig. 5(a), dashed edges denote new dependencies, and dotted edges represent advice relations.