# Using a Low-Level Virtual Machine to Improve Dynamic Aspect Support in Operating System Kernels

Michael Engel
Dept. of Mathematics and Computer Science
University of Marburg
Hans-Meerwein-Str.
D-35032 Marburg, Germany
engel@informatik.uni-marburg.de

Bernd Freisleben
Dept. of Mathematics and Computer Science
University of Marburg
Hans-Meerwein-Str.
D-35032 Marburg, Germany
freisleb@informatik.uni-marburg.de

## ABSTRACT

Current implementations of software providing dynamic aspect functionality in operating system (OS) kernels are quite restricted in the possible joinpoint types for native code they are able to support. Most of the projects implementing advice for native code use basic technologies adopted from instrumentation methods which allow to provide *before*, *after* and *around* joinpoints for functions. More elaborate joinpoints, however, are not available since support for monitoring native code execution in current CPUs is very restricted without extensive extensions of the compiler toolchain. To realize improved ways of aspect activation in OS kernels, we present an architecture that provides an efficient low-level virtual machine running on top of a microkernel system in cooperation with an aspect deployment service to provide novel ways of aspect activation in kernel environments.

## 1. INTRODUCTION

Current implementations providing dynamic aspect functionality today typically are based on either modifications of a high-level virtual machine (VM) like the JVM or modifications of the instruction stream that is executed on demand on the machine code level. Systems based on high-level VMs, like e.g. Steamloom [2], provide a rich set of functionality and are able to supply rather complex joinpoint models since they have the ability to intercept the execution of virtual machine instructions. Due to general characteristics of these VMs – no pointers are available in Java, languages running on top of the VM (Java, C#) are not widely used as system implementation languages – providing aspect support for an operating system kernel running on top of the VM is not feasible[1]. In contrast, aspect activation directly on the machine instruction level, provided by systems like TOSKANA [10] and Arachne [17], is restricted to the amount of interaction that is possible by dynamically rewriting the instruction stream using the available symbolic information in the executable code.

[1]There are, however, operating systems written in Java. These have never become mainstream, though.

In this paper, `TOSKANA-VM`, a novel way to provide aspect support for legacy operating systems is presented. Based on experiences with the NetBSD-based implementation of TOSKANA using native code manipulation [10], an intermediate approach between direct control of the native instruction execution and employing a high-level virtual machine is chosen. Since the abstraction level of a virtual machine is required in order to gain improved control over the execution of instructions, an existing low-level machine, LLVM [14], was chosen as the basis for TOSKANA-VM. In addition to the virtual machine itself, LLVM consists of a complete compiler toolchain providing support for compiling C and C++ programs that use GNU extensions. This is the basis for running a kernel on top of LLVM with only a moderate amount of modifications.

LLVM, however, is not well suited to run as a virtualization layer on top of the bare hardware. Thus, TOSKANA-VM uses the L4 microkernel as basis for executing LLVM instances. Here, L4 provides only minimal kernel functionality like memory management, task and thread abstractions and a fast implementation of inter-process communication. On top of L4, a mix of LLVM instances with their associated programs in LLVM bytecodes and native code can run concurrently. One of the native programs running on top of L4 is the weaver, which is responsible for activating and deactivating joinpoint shadows in the particular LLVM instance using IPC notifications.

This paper is organized as follows. Section 2 describes the overall structure of a system based on the L4 microkernel and an operating system personality, followed by an overview of low-level virtual machines in section 3. Section 4 describes ways to implement aspect-oriented functionality in a virtual machine. An overview of the L4- and LLVM-based system structure is contained in section 5. Section 6 summarizes related work. Section 7 concludes the paper and outlines areas for further research.

## 2. MICROKERNEL-BASED SYSTEMS

Compared to traditional monolithic operating system kernels, a microkernel-based system divides the functionality it provides into several system components that are cleanly separated from each other. At the base of the system, the

.

microkernel itself provides only the absolute minimum functionality of a kernel – in the case of L4 used in this paper, this is restricted to memory management, task management and interprocess communication primitives. All other functionality usually contained in a monolithic kernel is delegated to so-called *kernel personalities*, which are essentially user-mode tasks running as microkernel applications.

## 2.1 L4 System Structure

The basis for all interaction with the hardware in a L4-based system [15] is the microkernel itself. L4 has complete control over the hardware and is the only process in the system running in privileged ("kernel" or supervisor) CPU mode. All other parts of the system run under control of the microkernel in non-privileged user mode. This includes all operating system personalities described in the next paragraph.

L4 is optimized for fast inter-process communication, which is extensively used throughout the system. IPC messages can be sent and received by any task in the system; messaging in L4 is synchronous, so the delivery of IPC information is guaranteed by L4 as soon as the IPC call returns to the caller. In addition, L4 supports a method to share address spaces between different tasks running on top of L4. Using the *flexpages* system, one task can share parts of its virtual address space with another task with page-sized granularity.

In order to support virtualization of OS instances, L4 provides abstractions for timers and interrupts as well as virtual memory management and encapsulates these as IPC messages.

## 2.2 OS Instances

Since L4 only provides basic kernel functionality, an additional layer of software is required to implement the features required by application programs that L4 is lacking. This is realized in the form of an *operating system personality* that provides the standard interfaces of a traditional monolithic OS kernel to applications running on top of it. In the case of L4, a port of Linux as a personality is available, called *L4Linux* [11]. L4Linux is a modified version of Linux 2.6 in which all critical hardware accesses (interrupt control, page table handling, timer control) are removed and replaced by IPC calls to the underlying L4 microkernel that performs these operations on behalf of the Linux personality (if permitted by the security guidelines).

L4Linux runs as an ordinary user mode process, thus several L4Linux instances are unable to interfere with each other. As a consequence, a virtualization on the level of L4Linux instances running in parallel is feasible and already used in several applications [19, 9].

## 3. THE LOW-LEVEL VIRTUAL MACHINE

LLVM is a virtual machine infrastructure consisting of a RISC-like virtual instruction set, a compilation strategy designed to enable effective program optimization during the lifetime of a program, a compiler infrastructure that provides C and C++ compilers based on the GNU compiler collection and a just-in-time compiler for several CPU architectures. LLVM does not implement things that one would expect from a high-level virtual machine. It does not require garbage collection or run-time code generation. Optional LLVM components can be used to build high-level virtual machines and other systems that need these services.

## 3.1 LLVM Instruction Set

The LLVM code representation is designed to be used in three different forms: as an in-memory compiler (providing the intermediate representation IR), as an on-disk bytecode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. This allows LLVM to provide a powerful intermediate representation for efficient compiler transformations and analysis, while providing a natural means to debug and visualize the transformations. All three different forms of LLVM code representation are equivalent.

The LLVM representation aims to be as light-weight and low-level as possible while being expressive, typed, and extensible at the same time. It aims to be a "universal IR", by being at a low enough level that high-level ideas may be cleanly mapped to it (similar to how microprocessors are "universal IRs", allowing many source languages to be mapped to them). By providing type information, LLVM can be used as the target of optimizations: for example, through pointer analysis, it can be proven that a C automatic variable is never accessed outside of the current function.

## 3.2 The LLVM Infrastructure

As fig. 1 illustrates, the LLVM compilation strategy exactly matches the standard compile-link-execute model of program development, with the addition of a runtime and offline optimizer. Unlike a traditional compiler, however, the .o files generated by an LLVM static compiler do not contain any machine code, but rather LLVM code in a compressed format.

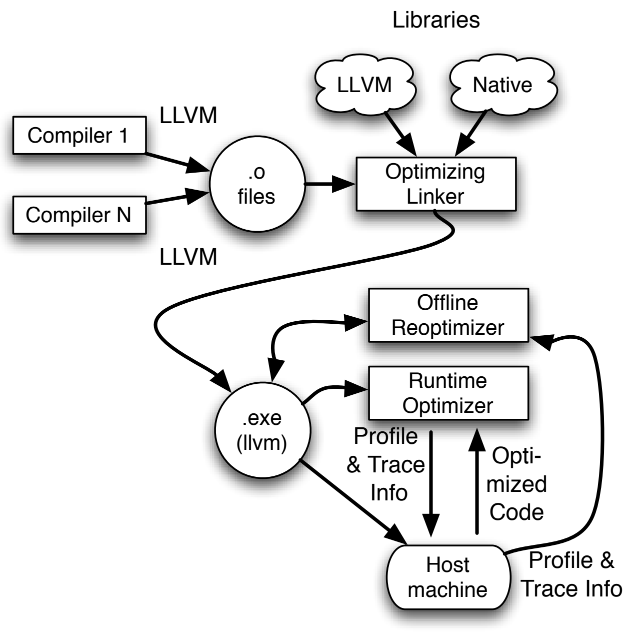

**Figure 1: LLVM Compilation Infrastructure**

The LLVM optimizing linker combines LLVM object files, applies interprocedural optimizations, generates native code, and links in libraries provided in native code form.

Once the executable has been produced, the developers (or end-users) of the application begin executing the program. Tracing and profiling information generated by its execution can be optionally used by the runtime optimizer to transparently tune the generated executable.

The system heavily depends on having a code representation with the following qualities:

The bytecode is high-level and expressive enough to permit high-level analyses and transformations at linktime and in the offline optimizer when profiling information is available. Without the ability to perform high-level transformations, the representation does not provide any advantages over optimizing machine code directly.

Also, the code has a dense representation, to avoid inflating native executables. Additionally, it is useful if the representation allows for random access to portions of the application code, allowing the runtime optimizer to avoid reading the entire application code into memory to do local transformations.

Finally, the code is low-level enough to perform lightweight transformations at runtime without too much overhead. If the code is sufficiently low-level, runtime code generation has low overhead in a broad variety of situations. A low-level representation is also useful because it allows many traditional optimizations to be implemented without difficulty.

## 4. ASPECT ACTIVATION IN A VM

When handling aspect activation in native code, the interception of the execution of arbitrary machine instructions is usually not supported extensively by standard CPUs. Consequently, the joinpoint model is quite restricted and dynamically inserting joinpoint shadows is an intricate task, since self-modifying code is used in the process.

Using a virtual machine instead of executing code directly on the CPU provides improved methods of detecting possible joinpoints, since all instructions are now either directly interpreted by the VM or translated into short native code segments by the just-in-time (JIT) compiler.

The following subsection describes the restricted "traditional" approach using code splicing, followed by a description of the advantages of the instruction-manipulating VM-based approach.

### 4.1 Code Splicing in Native Code

When directly working with native code, like in the TOSKANA project on NetBSD, the basic method for inserting dynamic joinpoint shadows into native code is *code splicing.* Code splicing is a technology that replaces the bit patterns of instructions in native code with a branch to a location outside of the predefined code flow, where additional instructions followed by the originally replaced instruction and a jump back to the instruction after the splicing location are inserted.

TOSKANA uses fine-grained code splicing, which is able to insert instrumentation code with the granularity of a single machine instruction. As shown in fig. 2, splicing replaces one or more machine instructions at a joinpoint with a jump instruction to the advice code with the effect that the advice code is executed before the replaced instruction.

This method has some significant drawbacks. When modifying native code, some complications show up that have to be taken care of in order to avoid corrupting activities currently running in kernel mode.

Since the execution of kernel functions may usually be interrupted at any time, it is desirable to make the splicing

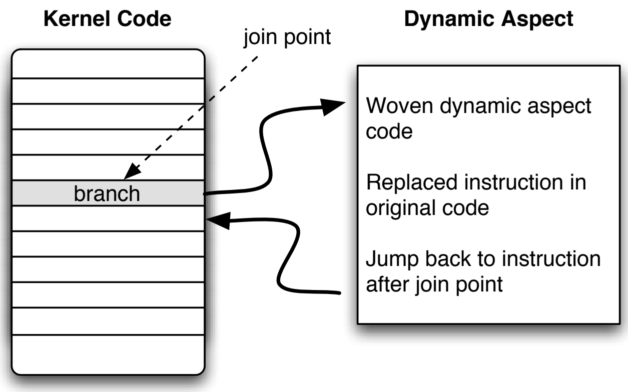
Kernel Code
join point
Dynamic Aspect
branch
Woven dynamic aspect code
Replaced instruction in original code
Jump back to instruction after join point

**Figure 2: Code Splicing**

operation atomic.

Another problem may be that more than one instruction has to be replaced by the jump instruction in the splicing process and the second of these replaced instructions is a branch target. A branch to that address would then end up trying to execute the bit pattern that is part of the target address as operation code with unpredictable results. In this case, either a workaround has to be activated – e.g. by rewriting an instruction before the joinpoint itself, thereby reducing the precision of the joinpoint location – or the replacement of an instruction must be avoided here, leading to a slightly restricted amount of available joinpoints.

The biggest problem with code splicing is that it only provides a very restricted joinpoint model. Since no information on register contents and values of pointers is available to the weaver, any operations involving dynamically calculated or loaded values are not eligible as possible joinpoint types.

Thus, essentially, only support for before, after, and around joinpoints is available by splicing in jumps to advice code at the beginning and return points of the respective function. A wider variety of joinpoint types is desirable, but can not be achieved using splicing.

### 4.2 Manipulation of Instruction Execution

Using LLVM gives the aspect weaver enhanced control over the execution of (VM bytecode) instructions. Instead of having to rely on self-modifying code at program runtime, the infrastructure executing the (byte-)code itself can now be instructed to intercept instruction execution, thereby providing much more detailed information about the current state of the machine.

Based on LLVM, TOSKANA-VM is able to supply a broader range of joinpoint types. The types currently implemented are described in the following paragraphs, accompanied with an explanation of the basic VM functionality executed to support them.

#### *Execution and Call*

LLVM provides two function call instructions, which abstract the calling conventions of the underlying machine, simplify program analysis, and provide support for exception handling. The simple call instruction takes a pointer to a function to call, as well as the arguments to pass (by value). Although all call instructions take a function pointer to invoke (and are thus seemingly indirect calls), for direct calls, the argument is a global constant (the address of a function). This common case can easily be identified simply by checking if the pointer is a global constant. The second function call instruction provided by LLVM is the invoke instruction, which is used for languages with exception handling.

When using code splicing, no clean distinction can be made between *execution* and *call* joinpoints, as the only point at which the weaver can be certain that the function in question was actually executed is within the code of the function itself.

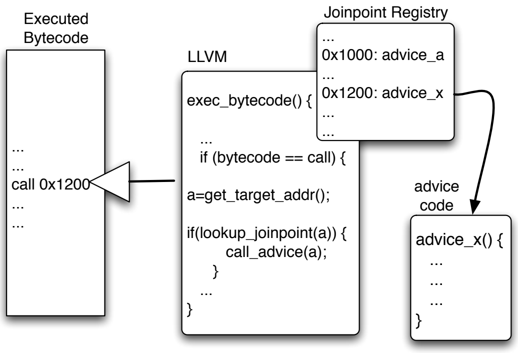

**Figure 3: A *Call* Joinpoint**

Implementing *call* joinpoints, as depicted in fig. 3, requires an interception of the call instruction described above. The target of the call instruction has to be compared against a list of active joinpoints and the related advice code is called appropriately.

*Execution* joinpoints – illustrated in fig. 4 – however, have a different semantics, since they exist at a point when the body of code for an actual method is executed. Thus, interception of the call is not sufficient; rather, the bytecode instruction pointer of the currently executing instruction has to be monitored. As soon as program execution enters (or leaves) the range of addresses defined by the function in question, advice code can be called. While monitoring the instruction pointer seems expensive at first look, the current implementation uses a fast hash-table mechanism to speed up the lookups. In future versions, annotated bytecodes could accelerate this functionality even further, requiring enhanced tool support.

### Variable Assignment and Access

Capturing variable assignment and access was not possible using splicing, since accesses to variables in native code not only occur using a direct address reference (which could be detected), but more commonly using pointers to variables contained in registers or calculated target addresses that were not available to the splicing process.

In LLVM, however, the final address of every read or write instruction executed by the VM is well-known. Hence, interception of read accesses (i.e., variable read) and write accesses (i.e., variable assignment) can be intercepted and corresponding advice code can be executed as the address of the variable is well-known from the symbol table.

A problem with joinpoints triggering on variable accesses lies in variables that are stored in registers for the sake of faster access times. Here, the VM provides a mapping from memory addresses to register contents; however, a tracking of variables in registers in the VM is necessary which is time-consuming. Thus, currently only *volatile* variables and variables to which an address operator (&) has been applied can be used as variable access joinpoints.

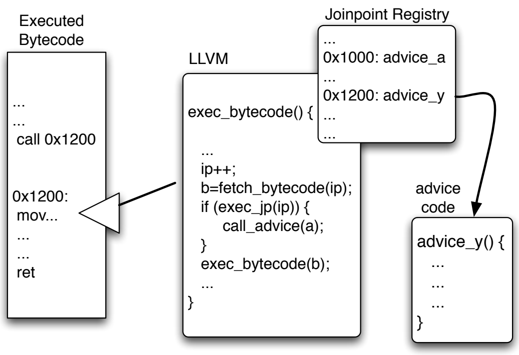

**Figure 4: An *Execution* Joinpoint**

## 5. SYSTEM STRUCTURE: L4+LLVM

An overview of the system is given in figure 5. On top of the microkernel, the infrastructure consists of one or more instances of LLVM running in their own address spaces and the weaver as a separate process that handles communication with the VM instances.

Running on top of L4, one or more instances of LLVM execute, these in turn can run L4Linux instances compiled to bytecode or other L4-based applications compiled with LLVM as their target.

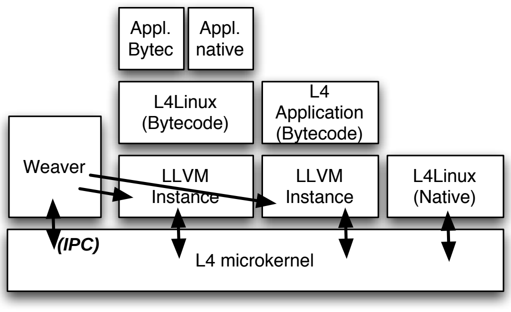

**Figure 5: L4/LLVM-based TOSKANA-VM System**

The dynamic aspect weaver is a separate component running on top of L4. It receives joinpoint description and weaving/unweaving requests from other tasks in the system and instructs a LLVM instance to add or remove the joinpoint description to respectively from its table of active joinpoints. Then, the weaver inserts the specified advice code in the address space of the LLVM instance in question using shared memory obtained via L4 flexpages.

In this L4-based system, tasks compiled to native code can execute in parallel to the LLVM instances. Of course, these tasks provide no support for dynamic aspects.

## 6. RELATED WORK

### 6.1 Steamloom

The Steamloom virtual machine [2] is an extension to an existing Java virtual machine. It follows the approach to tightly integrate support for AOP with the VM implementation itself instead of relying on a set of Java classes. Steamloom comprises changes to the underlying VM's object model and just-in-time compiler architecture. Its approach to dynamic weaving is to dynamically modify method bytecodes and to schedule those methods for JIT recompilation afterwards.

Steamloom was built to evaluate the benefits that can be gained by implementing AOP support in the execution layer instead of at application level. Various measurements [12] have shown that both flexibility and performance significantly benefit from such an integrative approach.

### 6.2 Arachne

Arachne [17] is a dynamic weaver for C programs. It uses the $\mu$Dyner AOP infrastructure for writing and dynamically deploying aspects into running user mode C applications without disturbing their service. The implementation of joinpoints, however, requires source instrumentation of the program to reduce the cost of dynamic weaving. $\mu$Diner provides a special *hookable* source-level annotation with which the developer of the base program annotates points in the program at which dynamic adaptation is permitted. Here, only functions and global variables can be declared to be hookable. The aspect code is written using a special extension of C that provides the syntax for specification of joinpoints and advice types.

### 6.3 a-kernel

Research in AOP in OS kernels was initiated by [7], where problems crosscutting a common layered operating system structure were identified using FreeBSD as an example. Based on AspectC, an AOP extension of C, the a-kernel project tries to evaluate the usability of aspects to improve OS modularity and reduce the complexity and fragility associated with the implementation of an operating system.

In [5], [8], [6], and [4], various cross-cutting concerns are implemented as static aspects using AspectC. In addition, an analysis of code evolution implementing cross-cutting concerns between different versions of FreeBSD is undertaken and the evolution is remodelled using static aspects.

Further development resulted in the RADAR [18], a low-level infrastructure using dynamic aspects in OS code. Currently, no experience with implementing the system seems to exist.

### 6.4 Singularity

Singularity [13] is a new research operating system developed by Microsoft focussing on the construction of dependable systems through innovation in the areas of systems, languages, and tools. Based on executing operating system code using a VM (MSIL) running on top of a microkernel, this system shows similarities to the approach presented in this paper. However, support for AOP is not mentioned in the related publications – a combination of dynamic AOP approaches for .NET and Singularity might be an interesting alternative to TOSKANA-VM.

### 6.5 DTrace

DTrace [3] is a toolkit developed by Sun Microsystems to dynamically insert instrumentation code into an unmodified, running Solaris OS kernel. Unlike other solutions for dynamic instrumentation that execute native instrumentation code, DTrace implements a simple virtual machine in kernel space that interprets byte code generated by a compiler for the "D" language, which is an extension of C specifically developed for writing instrumentation code.

D provides safe instrumentation of the kernel. To avoid endless loops in instrumentation code, only forward branches are permitted by the VM. Thus, the functionality of D programs is relatively restricted. While this provides a lot of security when dynamically inserting code into random spots in the kernel, the execution model provided by DTrace is too restricted to implement general advice code.

### 6.6 Xen

Xen [1] is a virtual machine monitor for x86 that supports execution of multiple guest operating systems with high levels of performance and resource isolation using *paravirtualization* technology. On top of Xen, different operating systems are able to run concurrently.

Xen requires modifications to kernels running on top of it, but applications run unmodified. Due to the paravirtualization, which only virtualizes and intercepts certain privileged instructions, Xen is not capable of interception instruction flow at arbitrary points in the code, so deploying advice code is not possible using this virtualization approach.

### 6.7 VVM

Instead of designing and implementing a new virtual machine for each application domain, the goal of VVM is to virtualize the virtual machine itself. VVM supports so-called "VMlets" that contain a specification of a virtual machine implemented using VVM.

No large-scale operating system has been developed to run on top of either of these virtual machines, however, so the feasibility of this approach still has to be determined.

### 6.8 z/VM

Another low level virtual machine that is explicitly used to virtualize a single physical machine into distinct partitions each running its own operating system instance is IBM's z/VM[16], used on z-Series mainframe systems. z/VM supports a large number of operating systems running in parallel on top of the virtual machine[2] and would be an ideal target for implementing dynamic aspect support in the execution layer.

## 7. CONCLUSIONS

This paper presented a novel approach to providing enhanced aspect-oriented programming technology in the context of an operating system kernel. First results show that using a low-level virtual machine as a thin layer above the hardware while relegating basic system functionality to a microkernel directly executing in native code on the CPU provides a reasonable architecture to provide and experiment with joinpoint models that implement novel concepts

[2]A test has shown that 40,000 parallel Linux instances are possible

or operate on a more fine-grained level than the joinpoints available through code splicing.

The current implementation of an aspect-enhanced LLVM and the weaver task on top of L4 is capable of running test programs written in C compiled as LLVM bytecodes. The next step is the port of a complete kernel personality (e.g. L4Linux) to run in bytecode on top of L4 and LLVM. This should be feasible, since L4Linux already provides a separate architecture component for L4, which subsequently has to be augmented with the necessary low-level adaptations and compile infrastructure changes for a LLVM target.

The performance of a complex system like a kernel personality running in bytecode instead of native code will be an interesting focus of optimization. The currently available simple test cases give the impression that running a kernel in bytecode will be feasible performance-wise, though no exact data is available at this time.

Future work also includes the provision of an improved security model. The method currently available in L4 to restrict communication between processes – “clans and chiefs” – does not provide sufficient control over which processes are permitted to communicate. This model will be replaced by a new security infrastructure of an upcoming L4 release which will be the basis for future TOSKANA-VM releases.

## 8. REFERENCES

[1] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, and R. Neugebauer. Xen and the Art of Virtualization. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[2] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. ”Virtual Machine Support for Dynamic Join Points”. In *”Proceedings of the Third International Conference on Aspect-Oriented Software Development (AOSD’04)”*, pages 83–92. ACM Press, 2004.

[3] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. ”Dynamic Instrumentation of Production Systems”. In *”Proceedings of the USENIX Annual Technical Conference”*, pages 15–28. USENIX, 2004.

[4] Y. Coady and G. Kiczales. ”Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code”. In *”Proceedings of the Second International Conference on Aspect-Oriented Software Development (AOSD’03)”*, pages 50–59. ACM Press, 2003.

[5] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. ”Structuring System Aspects: Using AOP to Improve OS Structure Modularity”. In *”Communications of the ACM, Volume 44, Issue 10”*, pages 79–82. ACM Press, 2001.

[6] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J. S. Ong, and S. Gudmundson. ”Exploring an Aspect-Oriented Approach to OS Code”. In *”Proceedings of the 4th Workshop on Object-Orientation and Operating Systems at the 15th European Conference on Object-Oriented Programming (ECOOP-OOOSW) ”*. Universidad de Oviedo, 2001.

[7] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, J. S. Ong, and S. Gudmundson. ”Position Summary: Aspect-Oriented System Structure”. In *”Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HOTOS-VIII)”*, page 166. IEEE Press, 2001.

[8] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. ”Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code”. In *”Proceedings of of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)”*, pages 88–98. ACM Press, 2001.

[9] M. Engel and B. Freisleben. Wireless Ad-Hoc Network Emulation Using Microkernel-Based Virtual Linux Systems. *Proceedings of EuroSIM 2004*, pages 198–203, 2004.

[10] M. Engel and B. Freisleben. Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects. *”Proceedings of the Fourth Conference on Aspect-Oriented Software Development (AOSD’05)”*, 2005 (to appear).

[11] H. Härtig, M. Hohmuth, and J. Wolter. ”Taming Linux”. In *Proceedings of the 5th Australasian Conference on Parallel and Real-Time Systems.* IEEE Press, September 1998.

[12] M. Haupt and M. Mezini. ”Micro-Measurements for Dynamic Aspect-Oriented Systems”. In M. Weske and P. Liggesmeyer, editors, *Proceedings of Net.ObjectDays*, volume 3263 of *LNCS*, pages 81–96. Springer Press, 2004.

[13] G. C. Hunt and J. R. Larus. Singularity Design Motivation. *Microsoft Technical Report TR-2004-105*, 2004.

[14] C. Lattner and V. Adve. ”LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In *”Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04), Palo Alto, California”*, pages 75–84. ACM Press, 2004.

[15] J. Liedtke. ”$\mu$-Kernels Must And Can Be Small”. In *Proceedings of the 5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA.* IEEE Press, October 1996.

[16] R. A. Mullen. ”z/VM: The Value of zSeries Virtualization Technology for Linux”. In *”Proceedings of the IBM z/VM, VSE and Linux on zSeries Technical Conference, Miami, Florida”*. IBM, 2002.

[17] M. Segura-Devillechaise, J.-M. Menaud, G. Muller, and J. Lawall. ”Web Cache Prefetching as an Aspect: Towards a Dynamic-Weaving Based Solution”. In *”Proceedings of the 2nd International Conference on Aspect-oriented Software Development”*, pages 110–119. ACM Press, 2003.

[18] O. Stampflee, C. Gibbs, and Y. Coady. ”RADAR: Really Low-Level Aspects for Dynamic Analysis and Reasoning”. In *”Proceedings of the ECOOP Workshop on Programming Languages and Operating Systems”*. ECOOP, 2004.

[19] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. ”Towards Scalable Multiprocessor Virtual Machines”. In *Proceedings of the 3rd Virtual Machine Research & Technology Symposium (VM’04).* IEEE Press, May 2004.