A Dynamic Aspect-oriented System for OS Kernels

Yoshisato Yanagisawa Kenichi Kourai Shigeru Chiba Rei Ishikawa

Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology

{yanagisawa,kourai,chiba,rei}@csg.is.titech.ac.jp

Abstract

We propose a dynamic aspect-oriented system for operating system (OS) kernels written in the C language. Unlike other similar systems, our system named *KLASY* allows the users to pointcut not only function calls but also member accesses to structures. This feature helps the developers who want to use aspects for profiling or debugging an OS kernel. To enable this, KLASY uses a modified C compiler for compiling an OS kernel. The modified compiler produces extended symbol information, which enables a dynamic weaver to find the memory addresses of join point shadows during runtime. Since a normal C compiler produces only limited symbol information, other dynamic aspect-oriented systems for C have been able to pointcut only function calls. We have implemented KLASY for Linux with the GNU C compiler. Our experiments revealed that KLASY achieves sufficient execution performance for practical use. Our case studies disclosed that KLASY is useful for real applications.

Categories and Subject Descriptors D [*3*]: 4

General Terms Languages

Keywords Aspect-oriented Programming, Operating System, Dynamic AOP, Profiling and Debugging, Linux

1. Introduction

Tool support is a key component of efficient software development. This is also true for the development of an operating system (OS) kernel. To analyze the behavior of a kernel, profile the execution performance, or fix a kernel bug, a kernel event logger or profiler will be a powerful tool.

This paper discusses the use of aspect-oriented programming (AOP) for debugging and profiling an OS kernel. It proposes our aspect-oriented system called *KLASY* (Kernel-Level Aspect-oriented SYstem), which helps kernel developers of Linux write an aspect for producing trace messages for performance profiling or debugging.

KLASY supports dynamic weaving of programs written in the C language. Dynamic weaving is a crucial feature for helping kernel development since compiling and rebooting an OS kernel takes long time. If the developers had to recompile and reboot an OS kernel whenever they modify aspects, for example, to measure the elapsed time of a different code section, they would not use the tool.

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.

KLASY has been implemented as a front end of Kerninst [28], which is a system for modifying the binary of a running Linux kernel. The users of Kerninst can insert "*hook*" code at an arbitrary memory address for invoking another function, which prints a trace message, for example. Although Kerninst provides basic mechanisms that KLASY needs, it provides only low-level abstraction, which is not convenient for profiling or debugging. The users must manually calculate the memory address where hook code should be inserted unless the address is the entry point of a function. They must also calculate the memory address of a local variable if they want to obtain the value of that variable in a function invoked by hook code.

In this paper, we propose that AOP gives good abstraction for kernel profiling and debugging. However, existing dynamic AOP systems for the C language do not support as powerful functionality as static AOP systems since compiled assembly code is far different from original source code. Their functionality is rather equal to that of Kerninst although their abstraction is more convenient. To address this problem, we have modified the gcc compiler so that it will produce extended symbol information. KLASY exploits this extended symbol information and thereby it provides pointcut designators for selecting not only function execution but also member accesses to structures. It also provides pointcut designators for accessing local variables and target structures. This paper also describes limitations of our approach.

It is important that a member access can be identified as a join point. Assume that developers want to investigate a performance bottleneck of network I/O. To do that, the developers must obtain time stamps at a number of places spread over the kernel. A naive approach is to define a pointcut to enumerate all functions where time stamps are recorded but this approach is error-prone even if a wild card is used. A much simpler approach is to define a pointcut for enumerating accesses to members of only a few structures related to network I/O. For example, network I/O functions in the Linux kernel commonly use the sk_buf structure. Other kernel subsystems such as a scheduler and a virtual file system also use such a particular data structure.

In the rest of this paper, Section 2 mentions existing technology for kernel profiling. Section 3 presents KLASY. Section 4 reports the results of our experiments and mentions overheads due to KLASY. Section 5 presents related work and Section 6 concludes this paper.

2. Kernel Profiling

OS kernels are never mature products. Improving them is still a hot topic [21, 30, 25, 19]. For example, developers of Linux and FreeBSD are still actively developing new schedulers. They are discussing techniques for reducing overheads of system calls. A new idea of prefetching algorithm was recently proposed [13]. Redhat Inc. has released the TUX web server, which uses a tuned OS kernel for HTTP service [24].

Developers who want to execute performance tuning of OS kernels need a good performance profiler for OS kernels. They need an appropriate tool set for identifying performance bottlenecks and eliminating them. A naive approach to do that is to manually modify the source code of the OS kernel so that it will include code fragments for measuring execution time of various code sections. This approach, however, needs modifying, recompiling, and rebooting a kernel whenever the code fragments for measuring execution time is changed. Note that recompiling and rebooting an OS kernel often take non-negligible time. Manually editing source code to insert and remove measuring code is error-prone. Developers might forget to remove a few code fragments and thereby make a serious trouble. When they remove measuring code, they might wrongly remove statements irrelevant to the measurement.

Another approach is to use a kernel profiler. Developers do not have to directly edit the source code of the kernel only for performance profiling. A typical profiler such as LKST [17] can produce a trace log that records when a kernel event specified at compile time occurs. It automatically edits source code to produce a trace log according to the users' specification. Unfortunately, the flexibility of this approach is low; the users must statically determine the kernel events that can be recorded in a trace log when they occurs. They cannot measure the execution time of code sections that were not specified at compile time. This is a serious problem because the code sections that they want to measure will change during a session of performance tuning.

2.1 Kerninst

An on-line kernel instrumentation tool such as Kerninst [28] solves this problem. It allows the users to modify a running OS kernel so that a given function will be called when the thread of control reaches the specified machine address. Since it directly replaces several machine instructions of the running kernel, the users do not have to reboot the kernel. This feature is significantly useful because unexpected performance behavior that developers want to investigate often occurs a long time after a kernel is booted. A reboot-less tool improves the productivity of kernel development.

However, since Kerninst was not a tool dedicated to kernel profiling, the abstraction provided by Kerninst is a quite low level. Kerninst provides only limited capability for kernel profiling. Thus using Kerninst for performance profiling is a complicated task.

Suppose that we want to print a log message with a time stamp when inode_change_ok() function is called. The log message includes the value of i_uid member of local variable inode declared in that function. If we use Kerninst, we first define a function for printing a log message:

```
void print_log() {
  struct timeval tv;
  void *ebp;
  int uid;
  __asm__ __volatile__("movl %%ebp, %0" : "=r"(ebp));
  uid = ((struct inode *)ebp[11])->i_uid;
  /* ebp[11] is inode */
  do_gettimeofday(&tv);
  printk("inode.i_uid: %d at %d.%ld\n",
         uid, tv.tv_sec, tv.tv_usec);
}
```

This function is compiled as a kernel module and dynamically loaded into the kernel address space. Note that the value of the local variable inode is obtained by ebp[11], where ebp holds the value of the ebp register when the inode_change_ok() function is running. 11 is the offset of inode in the stack frame. It must be manually calculated from the memory layout of a stack frame.

Then we use Kerninst to insert a machine instruction at the beginning of the inode_change_ok() function so that this print_log() function will be invoked.

```
kapi_manager kmgr;
void insert_hook() {
    kapi_module kmod;
    kapi_function ifunc, pfunc;
    kapi_vector<kapi_point> entries;
    kapi_vector<kapi_snippet> args;
    kapi_call_expr hook;

    kmgr.attach("localhost", 32770);
    kmgr.findModule("kernel", &kmod);
    kmod.findFunction("inode_change_ok", &ifunc);
    ifunc.findEntryPoint(&entries);
    kmgr.findModule("profiler", &kmod);
    kmod.findFunction("print_log", &pfunc);
    hook = kapi_call_expr(pfunc.getEntryAddr(), args);
    kmgr.insertSnippet(hook, entries[0]);
}
```

This program is compiled and run as a user process. It first connects to the Kerninst manager process and requests to modify the kernel code. Then it finds the addresses of the entry points of the two functions inode_change_ok() and print_log(). Finally, it inserts a code snippet "*hook*" so that inode_change_ok() will first call print_log().

The program above looks simple because Kerninst provides a mechanism for finding the address of the entry point of a function. However, if we want to print a log message in the middle of that function, we must manually calculate the address where the *hook* code is inserted and then we must pass that address to insertSnippet() instead of entries[0]. Kerninst only helps us find the addresses of the entry and exit points of a function. It can also report the entry address of every basic block but we have to manually identify which statements in a source file each basic block corresponds to.

2.2 Aspect-oriented programming

Although Kerninst provides basic mechanisms for kernel profiling, it is a general-purpose system for extending a running kernel and thus the functionality of Kerninst is not sufficient for profiling. The most serious problem is that we must know how kernel source code is compiled into machine instructions; we must manually calculate an address where we want to print a log message and where the value of a variable we want to inspect is stored. We need a front-end system of Kerninst that provides higher-level abstraction than Kerninst.

We propose to use aspect-oriented programming (AOP) as such a higher-level abstraction since it is widely known as an excellent paradigm for logging and profiling. If we use an AOP system, profiling code can be described as a module separated from the kernel source files. That separate module is called *aspects*. An aspect consists of *pointcuts* and *advices*. An advice is a language construct similar to a function. It is invoked when the thread of control reaches execution points specified by a pointcut. A pointcut is a composition of several predicates; it selects execution points that match those predicates. Those execution points are also called *join points*. To bind an advice with a target program at the execution points specified by a pointcut is called *weaving*. It is normally part of a compilation process or a program-transformation process after compilation.

Although AOP is a good paradigm, existing AOP systems are not satisfactory. First, an AOP system we need is a dynamic AOP system, which can dynamically weave an aspect. As Kerninst does, the AOP system must be able to attach and detach a profiling aspect to a running kernel without rebooting it. There are several dynamic AOP systems for the C language but they support only limited kinds of join points. They do not provide significantly better functionality than Kerninst. For example, TOSKANA [11] allows users to select only function execution as join points. Arachne [10]

allows users to select not only function calls but also accesses to global variables and arbitrary memory blocks. However, any of them does not enable selecting member accesses to structures.

In OS kernels, a number of structures are passed to transfer a collection of data between functions. To trace such a data flow, developers must be able to select member accesses by pointcuts. Tracing how transferred data is used is not a simple task if only function execution can be selected as join points. Moreover, structures are often used to implement a polymorphism-like mechanism in OS kernels. They are used as substitutes for classes available in C++ and Java. Some members of structures are function pointers to a *method*, that is, a function specialized for a particular type of structure. The network I/O system, the virtual file system, device drivers, and so on are implemented with this technique. Therefore, developers would want to use pointcuts selecting member accesses to such function pointers so that they can trace a call graph. A similar result could be obtained by describing pointcuts that select all possible function executions belonging to the call graph. However, selecting member accesses to the structures used in a target subsystem is simpler and easier.

Another limitation of existing dynamic AOP systems for the C language is a mechanism for context exposure. AOP systems have a mechanism for passing context information of a join point to an advice. For example, if a join point is function execution, function arguments can be passed to an advice. Arachne [10] allows an advice to access a return value and global variables as well. However, existing AOP systems for the C language do not allow an advice to local variables visible at a join point. Exposing local variables at a join point to an advice might not be an appropriate design with respect to modularization but it is often necessary for profiling.

3. KLASY: Kernel Level Aspect-oriented System

As we have shown above, existing AOP systems are not appropriate for profiling and debugging OS kernels. We propose our dynamic AOP system named KLASY (Kernel Level Aspect-oriented SYstem) for profiling and debugging OS kernels written in the C language. Unlike other similar systems, KLASY enables pointcutting member accesses to structures and provides better accessibility to context information at join points. KLASY thereby provides better usability than Kerninst; the users of KLASY do not have to manually calculate memory addresses.

An aspect for KLASY is written in C although it includes XML-like tags. Figure 1 is an example program. It can be compiled and woven by the klasy command at any time during runtime. For example,

```
% klasy weave inode_trace.klasy
```

this weaves the aspect in the file inode_trace.klasy into the OS kernel. After this aspect is woven, it prints a log message when the i_uid member of the inode structure is accessed within the function body of inode_change_ok(). Unlike the example in Section 2.1, this aspect prints a log message not when the execution of inode_change_ok() starts but whenever the i_uid member is accessed in inode_change_ok().

An aspect is surrounded by the aspect tag. The import tag specifies the header files that are necessary to compile an advice body. When an advice body is compiled, two header files linux/kernel.h and linux/module.h are implicitly included. Other header files can be included by using the import tag.

The advice body is a code fragment written in the C language and it is surrounded by the before (or after) tag. KLASY does not support an around advice but it is not a serious drawback since an aspect in KLASY is for profiling and it rarely needs around advice,

```
<aspect>
  <import>linux/time.h</import>
  <advice>
    <pointcut>
      access(inode.i_uid) AND
      within_function(inode_change_ok) AND
      target(inode_value);
    </pointcut>
    <before>
      struct inode *i = (struct inode *)inode_value;
      struct timeval tv;
      do_gettimeofday(&amp;tv);
      printk("inode.i_uid: %d at %d.%ld\n",
             i-&gt;uid, tv.tv_sec, tv.tv_usec);
    </before>
  </advice>
</aspect>
```

Figure 1. An aspect written in KLASY (inode_trace.klasy)

which is mainly used to execute a function at a join point only if necessary. In an advice body, several characters must be escaped since an aspect is tagged in XML. For example, angle brackets (< and >) must be replaced with < and >, respectively. The ampersand & must be &. In an advice body, a special variable pc is available. It represents the current value of the program counter, that is, the memory address of the machine instruction selected by a pointcut.

The advice body follows a pointcut definition surrounded by the pointcut tag. KLASY currently provides seven pointcut designators: execution, access, within_file, within_function, target, local_var, and argument. The execution pointcut identifies function executions as join points. access identify both read and write member accesses as join points. Wild cards % are available in these pointcuts. Since * represents a pointer type in the C language, wild cards are not * but %.

If a join point selected by a pointcut is a member access, the target pointcut is available. It sets a given variable to a pointer to the structure that is accessed at the join point. In Figure 1, the pointer to the inode structure is bound to a variable inode_value that is available in an advice body. The type of inode_value is void*. If the within_function pointcut is used, the local_var pointcut is used for obtaining the value of a local variable at a join point. The within_file and within_function pointcuts select join points included in a specified file and function body, respectively. The argument pointcut is used for obtaining the argument value of a function specified with the execution pointcut. Multiple pointcuts can be composed by AND or OR operators. If within_function is composed by AND with another pointcut such as access, the selected join points are ones that satisfy the conditions specified by both within_function and access.

3.1 Source-based binary-level dynamic weaving

KLASY enables developers to dynamically weave an aspect into the Linux kernel without rebooting the OS. To provide a pointcut for selecting member accesses as join points, KLASY uses our new technique named *source-based binary-level dynamic weaving*, in which the kernel source code is compiled by an extended C compiler of KLASY and thereby a richer symbol information is produced. An aspect is dynamically woven in the compiled binary, that is, a running OS kernel by exploiting that richer symbol information. KLASY modifies the compiled binary of the running kernel.

3.1.1 Extended symbol information

To enable the access pointcut shown above, the target OS kernel must be compiled by our extended C compiler so that the compiled binary will include the symbol information that is necessary to lo-

cate all occurrences of member accesses to structures. We extended the GNU C compiler (gcc) to develop that C compiler.

To record the locations of member accesses, we extended the parser of gcc. Since the global variables lineno and input_filename represent the current line number and the file name during parsing, our extended parser records the values of those global variables as well as a member name and a structure name whenever it encounters a member accesses. Note that an abstract syntax tree produced by the gcc parser does not include type names. All type names are converted into integer identifiers. Hence our extended parser maintains a mapping from the integer identifiers to the type names.

To locate the memory addresses of the machine instructions corresponding to the join point shadow [20] of member accesses, KLASY also needs to know where the compiled code of a given source line is placed in memory. Although the original gcc produces such address information if it runs with a debug option -g, the produced information is not sufficient for KLASY. For example, consider pointcut accesses to the addr_limit member of the thread_info structure. A source file acct.c of the Linux 2.6.10 kernel includes an access to that member at line 493:

```
493: fs = get_fs();
```

Here, get_fs is a macro. The definition of this macro is at line 32 in uaccess.h:

```
32: #define get_fs()    (current_thread_info()->addr _limit)
```

Since this macro includes the member access, the line 493 in acct.c is selected by the pointcut. KLASY records this line number. Note that the line 32 in uaccess.h is not recorded. Since uaccess.h is not a compilation unit (uaccess.h cannot be compiled into uaccess.o) but included in other files, KLASY cannot easily find which object file (.o file) contains the machine instructions corresponding to the line 32 in uaccess.h. Thus KLASY records the line 493 in acct.c as a join point since this line should be contained in acct.o, which is obtained by compiling acct.c. However, according to the address information produced by the -g option, the machine instructions for the member access corresponds to the line 32 in uaccess.h. It does not correspond to the line 493 in acct.c. Although this design is suitable for debuggers, KLASY cannot find the machine instructions for the member access at line 493 in acct.c. To solve this problem, we modified gcc. The parser of gcc associates the source code after macro expansion:

```
fs = (current_thread_info()->addr_limit);
```

with both the line 493 in acct.c and the line 32 in uaccess.h. These two line numbers associated with the code is removed by the RTL (register transfer language) generator and only the line 32 in uaccess.h is associated after that. We modified the RTL generator and the sub systems following the RTL generator, such as an RTL optimizer and an assembler (gas), so that they can maintain multiple line numbers. [1] Also, KLASY can maintain multiple line numbers for inline functions.

[1] Some readers might think that such complex implementation is unnecessary if all the source files of the Linux kernel are preprocessed in advance by the cpp command with the -P option. This option suppresses generating #line directives, which represent line numbers before preprocessing. However, this approach loses the information of the original line numbers and thus KLASY could not generate a helpful warning message including a line number. Furthermore, the users would want to know the original line numbers of the selected join points. Another problem of the cpp -P approach is that it cannot maintain correct line numbers if two source lines are merged into one for optimization.

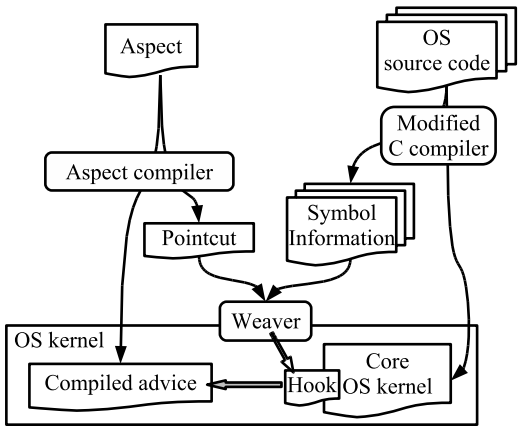

Figure 2. Overview of KLASY

3.1.2 Dynamic weaving

An aspect is compiled by the aspect compiler of KLASY. The advice bodies in the aspect are compiled into a loadable kernel module. It is loaded in the kernel space by the insmod command of Linux. Then KLASY resolves the memory addresses of the join points selected by pointcuts and KLASY inserts *hook* code at those addresses by using Kerninst [28]. The hook code calls an advice body when it is executed. The overall architecture is illustrated in Figure 2.

KLASY resolves the memory address of a join point by analyzing the symbol information produced by our extended gcc compiler. It first searches the symbol information for the file name and the line number of each join point selected by a given pointcut. Then it resolves the memory address corresponding to that line number. Our extended compiler compiles the OS kernel with the -g option and stores the extended address information in the debug information section of the compiled binary code. The compiler uses the DWARF 2 format to construct the binary code. KLASY first reads the .debug_info section of the binary of the kernel and finds the address information of the compilation unit that corresponds to the file name. Since the address information consists of .debug_line sections, KLASY reads them to find the memory address that corresponds the line number.

The minimum resolution of KLASY is a line since the symbol information does not include the exact memory address that corresponds to an expression such as member accesses. Therefore, for example, a before advice is executed not just before a join point such as a member access but before the line including the join point (shadow [20]) is executed. Some readers may think that KLASY can pick the exact memory address by using the information of the variable position pointcutted, which is in the symbol information. However, we believe that this approach should often fail because the variable position told in the symbol information is only warranted just before the line is executed. Moreover, this approach spent so much time for weaving that the developers would avoid using this tool. This limitation would not be a serious problem if KLASY is used for debugging and profiling an OS kernel because that resolution is the same as that of source-level debuggers and typical profilers.

Since the compiler may merge multiple lines for optimization, some lines including selected join points may disappear from the compiled binary. KLASY cannot find the memory address of those lines. However, in that case, KLASY tries to find the memory address of the line nearest to the original line. KLASY increments and decrements the line number one by one and tries to find the

memory address of the line. When KLASY finds the memory addresses of the two lines before and after the original line, it examines whether or not the two lines are in the same basic block. If they are, KLASY adopts the memory address of the line before (or after) the original line for a before (or after) advice. We assume that, in this situation, the original line is also in the same block and the original line is surely executed between them. Otherwise, KLASY prints an warning message and ignores the join points included in the original line. The basic blocks are computed by Kerninst.

If the memory address of a join point is found, KLASY inserts hook code at that address by using Kerninst. Since an advice body is transformed into a C function in a loadable kernel module, the hook code calls that function with the address of the join point as an argument. Kerninst substitute a jump instruction for the original instructions at that address. The jump instruction jumps to the hook code given by KLASY, which is placed somewhere else, and the original instructions are executed after the hook code. If the size of the replaced instruction is too small to put the jump instruction (5 bytes), Kerninst uses a breakpoint-trap instruction, which is only one byte. If the thread of control reaches that instruction, a trap handler is invoked and it executes the hook code and the instruction replaced with the breakpoint-trap instruction. Since the breakpoint-trap instruction causes software interruption and hence it implies a larger performance penalty than the jump instruction, Kerninst uses the breakpoint-trap instruction only when the jump instruction cannot be substituted for the original instruction. To weave multiple advice bodies at the same join point, KLASY generates a wrapper function that calls multiple advice bodies in turn. The hook code calls that function instead of advice bodies. This implementation is due to limitations of the current version of Kerninst.

3.1.3 Context exposure

The local_var, target, and argument pointcut designators pass the value of a local variable, an accessed structure, or function arguments, to an advice body; they can pass a value of execution context at a join point. For implementing this feature, KLASY generates a trampoline function, which bridges between hook code inserted by Kerninst and an advice body. When a trampoline function is called by hook code, it obtains a value of execution contexts and passes it to an advice body as an argument.

To obtain the value of a local variable or a function argument, KLASY reads the .debug_info section of the kernel binary to know the register number or the memory address of that variable. The .debug_info section is generated by the -g option to the gcc compiler. The -fno-omit-frame-pointer option is also used to remain a frame pointer to know an accurate position of a local variables. If a register is allocated to a variable, it is saved by Kerninst on stack memory before a trampoline function is called and thus KLASY reads the stack memory to obtain the value of such a register variable. On the other hand, if a stack frame is allocated to a variable, KLASY reads the saved value of ebp register to obtain the address of that stack frame and computes the memory address for a variable based on that.

If a pointcut includes access and target, a trampoline function must obtain the address of an accessed structure specified by access. If that structure is referred to by a local variable, KLASY obtains the value of that local variable by the way mentioned above and computes the address. For example,

```
inode.length
inode_ptr->length
```

If these accesses are selected by a pointcut, the addresses of the target structures are the values of &inode and inode_ptr. Our extended gcc compiler generates an extra symbol information of how those values are computed from local variable inode and inode_ptr. KLASY computes the addresses according to that symbol information.

```
<aspect>
  <advice>
    <pointcut>execution(check_free_space())
    </pointcut>
    <before>printk("execution of check_free_space");
    </before>
  </advice>
</aspect>
```

Figure 3. An example of execution pointcut

The target structure can be indirectly pointed to. Suppose that p is a local variable.

```
p->thread->fs
```

If the member access to fs is selected by a pointcut, the address of the target structure is p->thread. Our compiler also generates a symbol information of how the address is computed. A local variable and an intermediate member can be an array type. For example, our compiler generates a symbol information for the following access:

```
p[0]->threads[1]->fs
```

However, our compiler does not generate such a symbol information if an access is something like this:

```
current_thread_info()->exec_domain
```

since current_thread_info() is a function call. Our compiler generates a symbol information for computing a target address only if the address is a member of a structure directly or indirectly referred to by a local variable. If the target address is computed by an expression that KLASY does not support, our compiler reports an error.

3.2 Execution pointcut

KLASY also support the execution pointcut designator, which select function execution as a join point. For example, Figure 3 is an aspect that prints a trace message when a function check_free_space is executed.

The implementation of the execution pointcut is simple because the symbol information generated by a normal C compiler (in our case, gcc) includes the memory address of function entry points. KLASY uses Kerninst for inserting hook code, which calls an advice body when executed. If an advice is a before advice, KLASY inserts the hook code at the entry point of the specified function, that is, at the beginning of the function body. If an advice is an after advice, KLASY inserts the hook code at the exit point of the function. This insertion is processed by Kerninst. Due to the limitations of Kerninst, the execution pointcut cannot be used with a static function or an inlined function.

3.3 Unweaving

KLASY supports unweaving an aspect during runtime. KLASY records all the aspects that have been woven and, when the users request KLASY to unweave one of the aspects, KLASY removes the hook code inserted for that aspect, or removes the advice bodies from the trampoline functions. The modification of the binary

code is processed by Kerninst. The users can run a command for unweaving an aspect with the name specified by a command-line argument. For example,

```
% klasy unweave inode_trace.klasy
```

this command unweaves an aspect named inode_trace.

4. Experiment

We have developed a prototype of KLASY for the Linux 2.6.10 kernel (Fedora Core 2) with Kerninst 2.1.1 and gcc 3.3.3. This section reports the results of our experiments with this prototype. The machine we used for the experiments has an AMD Athlon™ XP 2200+ processor (1.8GHz), 1GB memory and an Intel® PRO/1000 network card.

4.1 Micro benchmark

First of all, we measured the overheads of a null advice. Since KLASY uses Kerninst as a back end, an advice is invoked by either jump instruction or breakpoint-trap instruction. If the size of the machine instruction at a join point (shadow) is too short, the breakpoint-trap instruction is used. Otherwise, the jump instruction is used. Furthermore, KLASY generate trampoline functions for either local_var or target pointcut is used. Therefore, we examined all these combinations. We implemented a simple new system call and wove a null advice with the kernel function implementing that system call.

Table 1 lists the results. We examined three cases: no trampoline function, a trampoline function obtaining data located in a register, and a trampoline function obtaining data located in a stack frame. If an advice is invoked through the jump instruction, an average overhead is about 16 nano seconds per join point. If it is invoked through the breakpoint-trap instruction, an average overhead is about 200 nano seconds. The overhead due to a trampoline function was negligible compared to the overhead of an advice invocation.

		trampoline	
	no	register	stack frame
jump	16	18	19
breakpoint trap	200	202	203

Table 1. Overheads of null advice (nano sec.)

4.2 Userland benchmarks

To evaluate overheads of KLASY in more realistic situations, we ran benchmark programs from UnixBench [1]. The programs are dhry2reg (Dhrystone 2 benchmark using register variables), whet (Whetstone benchmark), execl (performance of exec system calls), pipe (throughput of process pipes), context (performance of context switching between processes connected through a pipe), file1 (file copy 256 byte), file2 (file copy 1024 byte), file3 (file copy 4096 byte). create (process creation), shell (shell scripts), and syscall (overheads of system calls).

We first measured the execution performance of three Linux kernels with these benchmark programs. One is monolithic, which is a kernel compiled by the regular gcc and statically linked. Another is KLASY, which is a kernel compiled by our extended gcc for dynamic weaving. Any aspects were not woven during experiments. This kernel is also statically linked because KLASY doesn't support a dynamically-linked kernel. The last one is normal, which is a normal Linux kernel included in the Fedora Core 2 distribution. Note that the normal kernel is not statically linked. A number of kernel modules will be loaded during runtime. On the other hand, the monolithic and KLASY kernels are statically linked and thus they should have performance advantages.

Table 2 lists the index numbers reported by the benchmark programs. A larger number is better. The number in a parentheses shows an overhead to the index of the monolithic kernel. The number after "±" shows an error. According to this table, there is little difference among three kernels. Although the KLASY kernel is little slower than the monolithic kernel in several experiments, total amount of the overheads is only 4% with respect to their mean value. The KLASY compiler could not optimize as well as the regular compiler because our extended compiler must run with -g and -fno-omit-frame-pointer options as well as -Os option (normal optimization for a kernel) to obtain correct debug information. Those options disable a few optimization.

Then we ran several CPU-intensive benchmark programs (dhry2reg, syscall, pipe, execl, and context) with four kinds of aspects. Two aspects pointcut accesses to the nr_switches member of the runqueue structure and the other two pointcut accesses to the state member of the task_struct structure. For each set of the two aspects, the advice body of one aspect increments a counter while that of the other aspect records the current time. The nr_switches member represents the number of context switches that have been done and the state member represents the process state such as running and sleeping. When we wove these aspects, the weaver inserted hook code at 2 execution points for the nr_switches member and 50 execution points for the state member.

Table 3 lists the index numbers reported by the benchmark programs. A larger number means better performance. The number in round brackets represents the overheads of aspects and the number in square brackets below represents the number of calls to an advice body during the program execution. The overheads due to advice execution vary among benchmark programs. They depend on how frequently advice is executed, which instruction (jump or breakpoint-trap) is used for executing advice, and the execution time of the advice body. However, we can see that the overheads are acceptable unless an advice body is excessively frequently executed.

4.3 Case study with real applications

We below show our case studies. Note that we use the access pointcut designator to identify a number of interesting join points. In the C language, enumerating member accesses for identifying join points is often easier than enumerating functions even if we use wild cards; unlike C++ or Java, the C language does not provide a grouping mechanism for functions, such as packages and classes. Furthermore, in the first case study, we obtain a target structure by the target pointcut designator and use it to avoid logging the time stamp of unnecessary events.

4.3.1 Network tracing

One of our initial motivation to develop KLASY was to find performance bottleneck of the network I/O sub-system under heavy workload. Thus we tried measuring the elapsed time at several points of the network I/O sub-system when we sent bulk data from a remote host using the scp command. From the result, we could find that one of possible root causes is process scheduling.

Figure 4 is an aspect we used for this measurement. This aspect pointcuts accesses to all the members of the sk_buff structure and sets arg0 to a pointer to that structure. local.h is a file we wrote. Since some kernel data structures are defined in not a .h file but a .c file, we copied those data structures into local.h and included it in the aspect. The advice first casts the value of target to sk_buff. Then, if the protocol is not ARP, the advice records the current time as well as the program counter pc. DO_RDTSC is a macro provided by KLASY. It executes the rdtsc (Read Time Stamp

	Total	dhry2reg	whet	execl
monolithic	540 ± 10 (2%)	380 ± 0 (0%)	180 ± 0 (0%)	770 ± 20 (3%)
KLASY	520 (4%) ± 20 (4%)	380 (0%) ± 0 (0%)	180 (0%) ± 0 (0%)	730 (5%) ± 30 (4%)
normal	540 (0%) ± 20 (4%)	380 (0%) ± 0 (0%)	180 (0%) ± 0 (0%)	740 (4%) ± 30 (4%)

	file1	file2	file3	create
monolithic	430 ± 30 (7%)	390 ± 20 (5%)	380 ± 10 (3%)	1000 ± 50 (5%)
KLASY	390 (10%) ± 40 (9%)	380 (3%) ± 20 (5%)	380 (0%) ± 10 (3%)	970 (3%) ± 40 (4%)
normal	440 (-1%) ± 50 (12%)	400 (-2%) ± 20 (5%)	390 (-2%) ± 10 (3%)	990 (1%) ± 40 (4%)

	pipe	context	shell	syscall
monolithic	730 ± 60 (8%)	700 ± 40 (6%)	800 ± 10 (1%)	830 ± 10 (1%)
KLASY	650 (12%) ± 40 (5%)	670 (4%) ± 40 (6%)	770 (4%) ± 10 (1%)	820 (1%) ± 10 (1%)
normal	720 (1%) ± 60 (8%)	780 (-9%) ± 50 (7%)	780 (3%) ± 10 (1%)	770 (8%) ± 60 (7%)

Table 2. The performance index of the Linux kernels

		runqueue.nr_switch		task_struct.state	
	no aspect	counter	time	counter	time
		[# of calls.]		[# of calls.]	
dhry2reg	380 ± 0 (0%)	380 (0%) ± 0 (0%)	380 (0%) ± 0 (0%)	380 (0%) ± 0 (0%)	380 (0%) ± 0 (0%)
		[1,475]		[136,716]	
syscall	810 ± 20 (2%)	800 (2%) ± 50 (6%)	730 (10%) ± 50 (6%)	750 (7%) ± 60 (7%)	800 (2%) ± 50 (6%)
		[1,464]		[136,912]	
pipe	540 ± 60 (11%)	530 (2%) ± 30 (6%)	520 (3%) ± 40 (7%)	520 (4%) ± 30 (6%)	500 (6%) ± 30 (6%)
		[1,521]		[136,698]	
execl	740 ± 20 (3%)	730 (1%) ± 20 (3%)	730 (1%) ± 20 (3%)	730 (2%) ± 30 (4%)	720 (3%) ± 10 (1%)
		[976]		[127,482]	
context	540 ± 30 (6%)	510 (4%) ± 20 (4%)	530 (1%) ± 20 (4%)	420 (22%) ± 10 (2%)	420 (21%) ± 20 (4%)
		[43,457,582]		[102,466,611]	

Table 3. Indexes of Unix benchmark

Counter) machine instruction for obtaining the current time. Its execution time is about 6 nano seconds. STORE_DATA is another macro provided by KLASY. It is used to record data in kernel memory, which can be read later from a user process.

When we wove this aspect, the weaver could successfully insert hook code at 2494 lines but failed at 297 lines. KLASY failed to resolve the memory address of join points at 70 lines and failed to obtain the value of target at 227 lines. We explored the reason for these failures. The former failure occurred when a joinpoint is in a conditional expression that consist of multiple lines. We modified KLASY for avoiding that failure in case of if and while statements. However, KLASY still fails for other cases. The latter failure occurred when the value of target was lost by optimization. We can avoid this failure if we don't use target. Both kinds of failures would be unacceptable if we used KLASY for extending the functionality of the Linux kernel. However, the target application of KLASY is profiling and debugging, which do not need precise selection of join points according to our experience. Moreover, if the user avoids using target, the failure is only 70 lines (about 2.5% of all joinpoints). Also note that KLASY prints a warning message if it fails to finds the memory address at which hook code should be inserted. The users can see where KLASY fails to insert hook code.

We invoked the scp command from a remote host after we wove the aspect. For each arrival of a network packet, we could measure the elapsed time from when the network device of the target host received a packet, at several points of the network sub-system. Table 4 shows the result of tracing network I/O. We selected only 11 points out of measured 74 points and two different traces due to the space limitation of the paper. If the target host receives a packet, the thread of control first passes through line 2773 in e1000_main.c. For both traces, it takes 14 to 15 micro seconds from this line to line 4355 in tcp_input.c. However, the elapsed time from this line to line 234 in datagram.c is largely different: 12 or 688 micro seconds. When we examined source code, we found that tcp_rcv_established puts sk_buff on a queue and skb_copy_data_iovec dequeues it. Since skb_copy_data_iovec is executed by a process, the time between these two lines depends on process scheduling.

4.3.2 Tracing process switching

In our previous study, we examined how frequently an OS kernel switches processes under heavy workload, and revealed that behavioral anomaly between light- and heavy-weight processes under heavy workload is due to the thread scheduling policy in Linux [16]. To investigate that, we measured a CPU time quantum consumed by each thread in the Tomcat web application server [4], but we had to modify the kernel source code of Linux by hand since the execution point where we wanted to record the time in the scheduler was a member access to a structure. The previous aspect-oriented systems similar to KLASY did not enable us to pointcut member accesses.

If we used KLASY, such measurement could be implemented without modifying the kernel source code. We show the aspect for that measurement in Figure 5. This aspect pointcuts accesses the timestamp member of the task_struct structure within function

function	file	line	packet 1	packet 2
e1000_rx_checksum	e1000_main.c	2773	0.00	0.00
netif_receive_skb	dev.c	1638	1.06	1.39
netif_rx	dev.c	1500	1.68	2.17
ip_rcv	ip_input.c	367	3.43	4.35
ip_local_deliver	ip_input.c	275	5.56	6.67
tcp_v4_rcv	tcp_ipv4.c	1741	6.84	8.05
tcp_rcv_established	tcp_input.c	4238	11.14	12.62
tcp_event_data_recv	tcp_input.c	554	13.36	14.54
tcp_rcv_established	tcp_input.c	4355	14.23	15.43
skb_copy_datagram_iovec	datagram.c	234	25.93	703.76
__kfree_skb	skbuff.c	225	27.14	707.25

Table 4. Tracing result of network I/O (partial)

```
<aspect>
  <import>linux/skbuff.h</import>
  <import>linux/netdevice.h</import>
  <import>linux/netlink.h</import>
  <import>net/gen_stats.h</import>
  <import>net/sock.h</import>
  <import>net/tcp.h</import>
  <import>local.h</import>
  <advice>
    <pointcut>
      access(sk_buff.%) AND target(arg0)
    </pointcut>
    <before>
        struct sk_buff *skb = (struct sk_buff *)arg0;
        unsigned long long timestamp;

        if (skb-&gt;protocol != ETH_P_ARP) {
          STORE_DATA($pc$);
          STORE_DATA(skb);
          DO_RDTSC(timestamp);
          STORE_DATA(timestamp);
        }
     </before>
  </advice>
</aspect>
```

Figure 4. Aspect for tracing network I/O

bodies defined in sched.c. The advice body stores the program counter, the process identifier, and the current time. When we wove this aspect, the weaver could successfully insert hook code at 10 lines.

We ran both light- and heavy-weight services on Tomcat after we wove the aspect. To compute CPU time quantums from the log recorded by the advice body, we selected log entries related to process switches (line 2682 in sched.c). Table 5 shows the distribution of CPU time quantums consumed by threads. This shows that there are two peaks and the second peak is between 10 and 12. This observation is the same as that obtained in our previous study [16]. To perform our previous study, a few more aspects are needed in addition to the aspect in Figure 5. Those aspects are not shown here due to limited space but writing them is as easy as writing the aspect in Figure 5.

To investigate the overhead of advice execution in a real application, we measured the throughput of Tomcat using the ApacheBench benchmark program [3]. Table 6 shows the throughputs (the number of processed requests per second) for light- and heavy-weight services with or without the aspect in Figure 5. In case of Tomcat, the overhead due to using aspect was negligible

```
<aspect>
  <import>linux/sched.h</import>
  <import>asm/page.h</import>
  <advice>
    <pointcut>
      access(task_struct.timestamp) AND
      within_file(sched.c) AND target(arg0)
    </pointcut>
    <before>
      struct task_struct *p =
        (struct task_struct *)arg0;
      unsigned long long timestamp;

      DO_RDTSC(timestamp);
      STORE_DATA($pc$);
      STORE_DATA(p-&gt;pid);
      STORE_DATA(timestamp);
    </before>
  </advice>
</aspect>
```

Figure 5. Aspect for tracing process switching

	light-weight service	heavy-weight service
without aspect	650.20	6.87
with aspect	645.58	6.84
overhead	0.7%	0.4%

Table 6. Throughputs of Tomcat (requests/sec)

even if an advice was executed whenever a process switch occurred.

5. Related Work

5.1 Dynamic aspect-oriented systems for C/C++

There have been several dynamic aspect-oriented systems for the C language. However, these systems do not enable developers to pointcut accesses to a member of a structure. As we have mentioned, pointcutting member accesses is a significant feature for profiling and debugging an OS kernel.

TOSKANA [11] is a dynamic aspect-oriented system for the NetBSD operating system kernel. As KLASY does, it dynamically modifies the compiled binary of the kernel for weaving an aspect. Since TOSKANA uses the symbol information produced by a normal C compiler, it cannot allow developers to pointcut member accesses. The developers can only select function execution as join

range (ms)	0–2	2–4	4–6	6–8	8–10	10–12	12–14	14–
frequency	10,481	1,537	2,174	125	136	709	127	30

Table 5. Distribution of CPU time quantums

points. This design decision would be because TOSKANA was developed for autonomic computing, in which a kernel module is automatically replaced to dynamically adapt the kernel. For this purpose, the developers would rarely need to pointcut member accesses.

TOSKANA-VM [12] is a system that allows developers to dynamically weave an aspect with the kernel. The approach of TOSKANA-VM is similar to the approach of Steamloom [5, 15], which is a custom Java virtual machine extended for enabling dynamic weaving. The kernel of TOSKANA-VM is compiled by a special compiler into virtual machine code, which is run on a virtual machine named LLVM. Since the virtual machine code contains rich symbol information, TOSKANA-VM allows developers to pointcut various kinds of join points such as reading and writing a variable. However, the kernel must run on a virtual machine and thus this approach cannot be used for profiling a kernel directly running on native hardware.

DAC++ [2] and TinyC^2 [31] are dynamic aspect-oriented systems for user processes written in the C++ language. For weaving an aspect, those systems modify the compiled binary of C++ programs during runtime. Since they use the symbol information produced by a normal compiler, they only support pointcuts for selecting function execution as join points. Member accesses to structures cannot be join points. Arachne [10] is a dynamic aspect-oriented system for user processes written in the C language. Although it also uses the symbol information produced by a normal compiler, the pointcut designators of Arachne cover not only function calls but also accesses to global variables and memory blocks allocated by `malloc`. However, Arachne does not provide a pointcut designator for member accesses. Furthermore, pointcutting accesses to memory blocks implies serious performance penalties since Arachne uses a page fault for detecting accesses to the memory block.

μDyner [26] is also a dynamic aspect-oriented system for user processes written in the C language. It is a predecessor of Arachne. μDyner inserts hook code at every join point marked as *hookable* when a source file is compiled. The hook code examines whether or not the join point is selected by a pointcut during runtime and, if it is selected, the hook code executes the associated advice. Although μDyner potentially can support various kinds of join points since it inserts hook code at compile time, the developers must annotate source files by the hookable mark so that the hook code will be inserted at appropriate join points. If a join point is not marked as *hookable*, it cannot be selected by a pointcut during runtime. Furthermore, overheads due to hook code are not negligible if the number of hookable join points is large.

5.2 Static aspect-oriented systems for C/C++

Since static aspect-oriented systems can utilize the complete source-level information of a program when weaving an aspect at compile time, they can easily provide various pointcut designators. Most of such source-level information is discarded after compilation and thus existing dynamic aspect-oriented systems that modify the compiled binary for weaving have not been able to utilize that information except limited symbol information. However, static aspect-oriented systems are not appropriate for profiling and debugging an OS kernel as we mentioned in Section 2. The developers should be able to weave a new aspect on demand without rebooting a kernel.

AspectC is an early static aspect-oriented language for the C language and it has been used for showing that aspect-oriented programming works well for modularizing an OS kernel [9, 7, 8]. For example, the mechanism for prefetching a disk block must cut across a virtual memory sub-system and a disk sub-system. To implementing that mechanism separately from those sub-systems, aspect-oriented programming is necessary.

AspectC++ [27] is another static aspect-oriented system for the C/C++ language. It provides the same kinds of pointcut designators that AspectJ does [18] since it is a source-to-source translator from an aspect-oriented language to the regular C/C++ language. Since it is a static aspect-oriented language, it is not appropriate to our applications.

5.3 Other related tools

There have been a few tools for modifying the binary code of a running OS kernels. Kerninst [28], which is the back-end of KLASY, is one of those tools. Unfortunately, their abstraction is not source-level but assembly-level. The users must directly deal with memory addresses of functions and global variables. This is a serious drawback of Kerninst for profiling and debugging an OS kernel since the developers should want to see source-level abstraction. Also, GILK [22] is a similar tool to Kerninst but it uses only `jmp` instruction to insert hook code. The performance of GILK is better than that of Kerninst but GILK supports only old Linux kernels.

SLIC [14] is another tool for modifying a running OS kernel. It only allows developers to change an entry of jump tables such as an interrupt vector table and to insert hook code at the beginning of a function. These ability is not sufficient for profiling and debugging an OS kernel.

LKST [17], DTrace [6], SystemTAP [23], and LTT [29] are tools for producing log messages about events occurring in the kernel. The users of these tools can dynamically control when a log message is produced. However, the execution points where a log message can be produced must be statically determined when the kernel is compiled. The users can only select some of the pre-fixed execution points and activate them to produce a log message.

6. Concluding Remarks

This paper presents KLASY, which is our dynamic aspect-oriented system for debugging or profiling the Linux kernel. We developed the *source-based binary-level dynamic weaving* technique for implementing KLASY and thus KLASY allows users to pointcut member accesses to structures. It is an important feature since selecting member accesses to a few structures related to a profiling is much simpler than selecting a large number of functions related to a profiling. In the C language, structures are often shared among functions implementing the same concern. They are units of modules as classes in Java and C++. It also provides pointcut designators for accessing local variables and target structures. Allowing accesses to local variables might be inappropriate with respect to modularization but KLASY is mainly for profiling and debugging. Since accessing local variables is necessary in those domains, we relaxed modularization concern.

Our dynamic weaving technique uses a modified C compiler that generates extended symbol information, which the dynamic weaver refers to for finding the memory addresses of the join points (shadow) selected by pointcuts. The dynamic weaver modifies the

binary code of a running kernel so that hook code for executing an advice body is embedded at those addresses. The extended symbol information also enables accesses to target structures.

The contributions of this paper are to present the source-based binary-level dynamic weaving and also to discuss limitations of that approach. One drawback of that approach is that our modified gcc compiler does not optimize as well as the original gcc since our compiler must generate extended symbol information. The performance penalty is about 0% to 30%. Another drawback is that an aspect weaver may fail to find some join points selected by a pointcut if a compiler performs serious code motion for optimization.

References

[1] Unixbench. `http://www.tux.org/pub/tux/niemi/unixbench/`.

[2] S. Almajali and T. Elrad. Coupling availability and efficiency for aspect oriented runtime weaving systems. In *Proceedings of the Second Dynamic Aspects Workshop (DAW05)*, march 2005.

[3] Apache HTTP Server Project. Apache HTTP server benchmarking tool. `http://httpd.apache.org/`.

[4] Apache Software Foundation. Apache Tomcat. `http://tomcat.apache.org/`.

[5] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[6] 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 Association, June 2004.

[7] Y. Coady and G. Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59. ACM Press, 2003.

[8] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong. Structuring operating system aspects. *Communications of the ACM (CACM)*, october 2001.

[9] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectC to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98. ACM Press, 2001.

[10] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[11] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 51–62, New York, NY, USA, 2005. ACM Press.

[12] M. Engel and B. Freisleben. Using a low-level virtual machine to improve dynamic aspect support in operating system kernels. In *Proceedings of the Fourth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, march 2005.

[13] K. Fraser and F. Chang. Operating system i/o speculation: How two invocations are faster than one. In *Proceedings of the USENIX Annual Technical Conference (General Track)*, pages 325–338. USENIX Association, June 2003.

[14] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the USENIX Annual Technical Conference (NO 98)*, june 1998.

[15] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An execution layer for aspect-oriented programming languages. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 142–152, New York, NY, USA, 2005. ACM Press.

[16] H. Hibino, K. Kourai, and S. Chiba. Difference of degradation schemes among operating systems. In *Proceedings of Workshop on Dependable Software - Tools and Methods*, Dependable Systems and Networks (DSN-2005), pages 172 – 179, June 2005.

[17] Hitachi, Ltd. and Fujitsu, Ltd. Linux kernel state tracer, 2001, 2005. `http://lkst.sourceforge.net/`.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[19] G. Lehey. Improving the FreeBSD smp implementation. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 155–164. USENIX Association, June 2001.

[20] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In *Proc. of Foundations of Aspect-Oriented Languages Workshop*, AOSD 2002, pages 17–26, 2002.

[21] S. Molloy and P. Honeyman. Scalable Linux scheduling. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 285–296. USENIX Association, June 2001.

[22] D. J. Pearce, P. H. J. Kelly, T. Field, and U. Harder. GILK: A dynamic instrumentation tool for the linux kernel. In *Computer Performance Evaluation / TOOLS*, pages 220–226, 2002.

[23] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium*, volume 2, pages 49–64, july 2005.

[24] Red Hat, Inc. Red hat content accelerator (tux), 2001, 2002. `http://www.redhat.com/docs/manuals/tux/`.

[25] J. Roberson. Ule: A modern scheduler for FreeBSD. In *Proceedings of BSDCon '03*, pages 17–28. USENIX Association, September 2003.

[26] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. L. Lawall. Web cache prefetching as an aspect: towards a dynamic-weaving based solution. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119. ACM Press, 2003.

[27] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.

[28] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.

[29] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the USENIX Annual Technical Conference*, pages 13–26, june 2000.

[30] S. Yamamura, A. Hirai, M. Sato, M. Yamamoto, A. Naruse, , and K. Kumon. Speeding up kernel scheduler by reducing cache misses - effects of cache coloring for a task structure -. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 275–286. USENIX Association, May 2002.

[31] C. Zhang. TinyC^2: Towards building a dynamic weaving aspect language for c. In *FOAL 2003 Proceedings Foundation of Aspect-Oriented Languages Workshop at AOSD 2003*, march 2003.