A Source-level Kernel Profiler based on Dynamic Aspect-Orientation

Yoshisato YANAGISAWA
Tokyo Institute of Technology
2-12-1-W8-50 Ohkayama, Meguro-ku
Tokyo 152-8552, JAPAN
yanagisawa@csg.is.titech.ac.jp

Shigeru CHIBA
Tokyo Institute of Technology
2-12-1-W8-50 Ohkayama, Meguro-ku
Tokyo 152-8552, JAPAN
chiba@acm.org

Kenichi KOURAI
Tokyo Institute of Technology
2-12-1-W8-50 Ohkayama, Meguro-ku
Tokyo 152-8552, JAPAN
kourai@csg.is.titech.ac.jp

ABSTRACT

We present a source-level kernel profiler named KLAS. Since this profiler is based on dynamic aspect-orientation, it allows the users to describe any code fragment in the C language. That code fragment is automatically executed for collecting detailed performance data at execution points specified by the users. Enabling dynamic aspect-orientation is crucial since otherwise the users would have to reboot an operating system kernel whenever they change aspects. Although KLAS dynamically transforms the binary of a running operating system kernel for weaving an aspect at runtime, unlike other similar tools, the KLAS users can specify those execution points, that is, joinpoints through a source-level view. For example, the users can describe a pointcut that picks up accesses to a member of a structure; they do not have to explicitly specify the addresses of the machine instructions corresponding to the member accesses. We have implemented this feature by extending a C compiler to produce augmented symbol information. KLAS has been implemented for the FreeBSD operating system with the GNU C compiler.

1. INTRODUCTION

During the history of operating systems (OS), performance tuning of OS kernels has been an important topic for kernel developers. Even nowadays, the kernel developers are making serious efforts to run the OS kernel as fast as possible. They are still improving scheduling algorithms, implementation of network stack, lock mechanism, and so on. For example, both Linux and FreeBSD recently introduced new implementation of their process schedulers.

Investigating a performance bottleneck is the important first step for improving the performance of OS kernels. To do this, using a sophisticated performance profiling tool for OS kernels is mandatory. Here "sophisticated" means that the profiler reports not only the number of calls to each function constituting an OS kernel but also more detailed data specified by the users. In fact, we are studying a performance bottleneck of network processing of the FreeBSD operating system since we observed inappropriate behavior of the network module when multiple processes are simultaneously executing network I/O operations. We need a profiler that allows us to produce a log message including time stamp at any execution points (a.k.a joinpoints in AOP) that we specify.

This paper presents our kernel profiler called *KLAS*. It is a dynamic aspect-oriented system and allows the users to execute a code fragment as advice at specified execution points in an OS kernel. The advice is normally used to record time stamps but it can also executing any code written in the C language, for example, printing a log message. Since KLAS is a dynamic aspect-oriented system, the users can dynamically weave an aspect with a running OS kernel. Enabling dynamic weaving is crucial since the users can avoid rebooting a OS kernel whenever they change an aspect during investigation of kernel performance. For KLAS, we have developed a new implementation technique for dynamic aspect-oriented systems. KLAS replaces machine instructions in an OS kernel with breakpoint-trap instructions so that advice is woven at the address of those instructions. To enable the users to specify pointcuts with a source-level view, such as accesses to a member of a structure, we modified a C compiler so that it will produce extra symbol information. KLAS uses this extra information to identify the machine instructions that correspond to the specified pointcut. We implemented KLAS for the FreeBSD operating system with the GNU C compiler.

The rest of this paper is organized as follows. Section 2 describes requirements for kernel profilers. Section 3 presents our new implementation technique for dynamic aspect oriented programing (AOP). It also shows an overview of the current implementation of KLAS. Section 4 compares KLAS and other systems, including AOP systems and non-AOP systems. We conclude this paper in section 5.

2. REQUIREMENTS

To investigate a performance bottleneck, using a performance profiling tool is mandatory; in particular, a tool that can measure the elapsed time between interesting execution points in the OS kernel is useful. However, existing tools or techniques do not satisfy our requirements for investigat-

ing kernel performance. Since modern OS kernels are implemented with object orientation in the C language, a number of interesting execution points are calls to functions specified by function pointers. That profiler does not support such execution points; it only supports functions statically resolved. We below mention our requirements for such a kernel profiler.

First, the kernel profiler must enable the users to measure elapsed time between given two execution points. The users must be able to give those execution points in the kernel at runtime and change them, if necessary, without rebooting the kernel. The ability to change the execution points is crucial. The users would first measure the execution time of a large code section and then they would gradually narrow the range of that code section to find a performance bottleneck. Since rebooting the whole kernel is a time consuming task, frequent rebooting significantly decreases our productivity. Rebooting also clears the whole memory image and thus the internal data of the network module. After rebooting, the behavior that the users want to investigate might disappear. Furthermore, the code snippet for measuring the elapsed time must be given by the users since the users may want to measure the elapsed time between the execution points in which a certain variable holds a specific value. To do this, the measurement code must check the runtime value of that variable but only the users can give such code depending on a particular use case. Also, the users may want to print a log message, for example, to record the value of an interesting variable.

Second, the profiler should support the C language. The users must be able to specify execution points by indicating a point in a source file. This is mainly because the FreeBSD operating system, and other major operating systems like Linux, are written in C. Several features of the C language makes it difficult to develop a kernel profiler. For example, the macro processor makes it difficult to specify an execution point and the compiled binary includes only limited symbol information.

Third, the execution points that the users can specify for profiling must be fine grained. The possible execution points must include not only function calls but also member accesses, that is, accesses to members of structures. A number of execution points that we are interested in for performance profiling are function calls through function pointers. Modern OS kernels use function pointers for inter-module function calls since function pointers can be used for implementing a kind of polymorphism in the C language. If the read or write system call is issued, the OS kernel invokes a function pointed to by a function pointer associated with the accessed I/O device. The function pointer associated with each I/O device points to the read/write function dedicated for that device. The VFS (Virtual File System) uses the same technique for dispatching to a function appropriate to each type of file system. The network module of FreeBSD and NetBSD, which are descendants of 4.3BSD, uses this technique for deallocating a memory buffer (mbuf) in a means depending on a network device.

Finally, the prove effects due to the profiling should be minimized. If the overheads of measuring elapsed time is large, the obtained data would be obviously inaccurate. Once necessary data are obtained, the profiling code for the time measurement must be removed to avoid disturbance of the kernel behavior while the elapsed time of a different code section is being measured.

A naive approach for performance profiling of OS kernels is to manually insert profiling code into source files of the kernel, compile the source files, and reboot the kernel. However, this approach is error-prone and does not satisfy our requirements since it needs rebooting.

3. KLAS: KERNEL LEVEL ASPECT-ORIENTED SYSTEM

To fulfill all our requirements, we have developed a new dynamic aspect-oriented system called KLAS (Kernel-level Aspect-oriented System) for FreeBSD 5.2.1. AOP is the most promising approach for our requirements. KLAS receives the definition of an aspect from the users through a KLAS command running in the userland. Then it dynamically patches the running OS kernel to weave that aspect into the kernel at runtime. Since KLAS uses a modified version of gcc for augmenting the symbol information contained in the compiled binary of the OS kernel, it allows the users to pointcut member accesses at the source-code level.

3.1 Overview of the KLAS system

KLAS is a dynamic aspect-oriented system for the OS kernel of FreeBSD. The users can dynamically weave an aspect into the running kernel so that they can change the code section of which they measure the execution time. They do not have to reboot the kernel when they change a woven aspect. This feature improves the efficiency of the users' investigation since they do not have to wait until the kernel is rebooted and the behavior that they want to investigate appears again. They can start investigation as soon as they find the behavior that they are interested in.

KLAS allows the users to pick out member accesses (accesses to a member of a structure) by pointcut. As we have already mentioned, it is a crucial feature that the users can specify that an advice body is executed when a particular member of function pointer type is accessed. For example, this feature helps us investigate a performance bottleneck of network processing since we can easily measure the execution time of functions accessing the mbuf structure.

An aspect definition for KLAS is described in XML. Figure 1 shows an example of an aspect definition for KLAS. It pointcuts accesses to the member ext_free of the m_ext structure. Since the value of ext_free is a function pointer, this member access is a function call. The advice body associated with this pointcut prints the current time and the arguments to the function when that member access is performed. In KLAS, special variables $eip, $ebp and $esp are available within advice body. They represent eip, ebp and esp register.

3.2 Implementation

KLAS inserts the *hook* code into the OS kernel for executing advice body when the thread of control reaches there. The overhead due to the hook code is minimum since KLAS

```
<aspect name="log_mbuf_clean">
  <pointcut>
    <member-access name="ext_free" struct="m_ext" />
  <pointcut>
  <before-advice>
    void* resolve_arg(long eip, long ebp, int argn)
    {
      /* resolve the N-th argument of
         ext_free function. */
    }
    struct timespec ts;
    nanotime(&ts);

    printf("mbuf_clean@%d,%lld, arg:0x%x,0x%x\n",
        ts.tv_sec, ts.tv_nsec,
        resolve_argument($eip,$ebp,1),
        resolve_argument($eip,$ebp,2));
  </before-advice>
</aspect>
```

Figure 1: Aspect Definition in KLAS

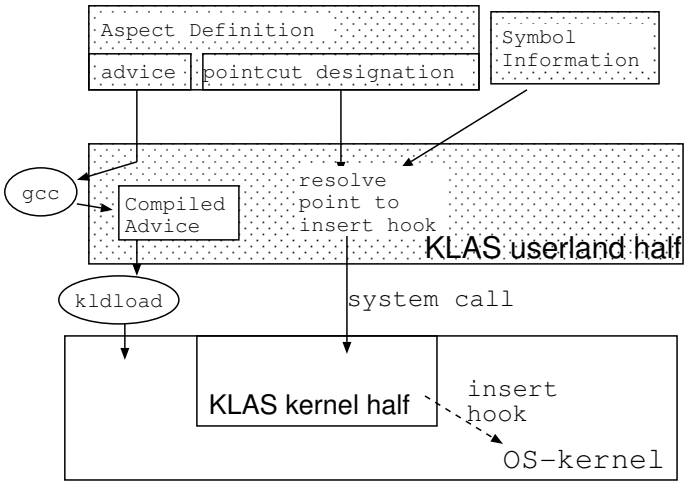

Figure 2: Implementation of KLAS

dynamically inserts the hook code only at the places corresponding to the joinpoint shadow picked out by given pointcuts. If the aspect is unwoven, the inserted hook code is also removed from the running OS kernel. Minimizing the overhead is important since the primary application of KLAS is to investigate a performance bottleneck in the OS kernel. If the overhead of using aspects is not negligible, the users may be confused by the disturbance by the prove effects and have a trouble to find a real performance bottleneck.

A unique feature of KLAS is that KLAS enables member accesses to be picked out by a pointcut. To do this, KLAS expands the symbol table contained in the compiled binary. This fine-grained pointcut helps the users to efficiently investigate a performance bottleneck in the OS kernel. The users can specify a pointcut to pick out interesting member accesses at the source-code level, and KLAS refers to the expanded symbol table so that it can insert the hook code at the machine instructions corresponding to those member accesses (Figure 2).

To use KLAS, the OS kernel must be compiled by our extended the GNU C compiler (gcc) with the -g debug option. During compilation, our compiler collects the names of structures and their members with the line numbers and the file names in which those members are accessed. The collected information is stored in an auxiliary file of the compiled kernel. Note that this information is not included in the normal symbol table of the compiled binary even if the -g option is given to the compiler. For example, the GNU C compiler discards this information after the parse tree is created; the structure names and the member names are converted from character strings to integer ID. numbers. The GNU C compiler uses not names but those ID. numbers for identifying structures and members after the parsing phase.

If KLAS is requested to dynamically weave a new aspect while the OS kernel is running, it refers the symbol information generated when the kernel was compiled. KLAS uses that information for identifying the addresses of the machine instructions corresponding to the joinpoints picked out by given pointcuts. To identify the address of a function, KLAS simply refers to the regular symbol table by invoking the nm command. To identify the address of a member access, KLAS performs the following three steps. First, KLAS refers to the auxiliary file generated by our extended compiler and obtains the file name and the line number at which that member access is executed. Then KLAS accesses the debug information, which is included in the regular symbol table. It uses the file name for identifying the name of compilation unit, which is an object file constituting the OS kernel, and it finally accesses the debug_line information (the DWARF2 format) of that compilation unit. The address of the line specified by the line number can be found in the debug_line information, which is also included in the regular symbol table. Since KLAS can obtain only the address of the first machine instruction of the line including the joinpoint, it cannot insert the hook code exactly at the instruction corresponding to that joinpoint. However, we believe that this limitation is not a serious problem for our application, which is investigating a performance bottleneck of the OS kernel. Moreover, this approach allows the users to use the same compiler that they are usually using for compiling the kernel because the information our modified GNU C Compiler generates is only a mapping between member accesses and line numbers.

KLAS uses the GNU C compiler (gcc) for compiling an advice body and the kldload command for loading the compiled advice body into the kernel land. After parsing an aspect definition written in XML, KLAS extracts an advice body and attaches the prologue and the epilogue to the advice body to make a source file of a loadable kernel module. This produced source file is compiled by gcc. The compiled binary is loaded by the kldload command. The advice body can be any code fragment if it is a valid C program in the kernel.

The loaded advice body is woven when a system call for dynamic weaving is issued. KLAS identifies the machine instruction corresponding to the joinpoint and replaces it with the breakpoint-trap instruction, which is the hook code of

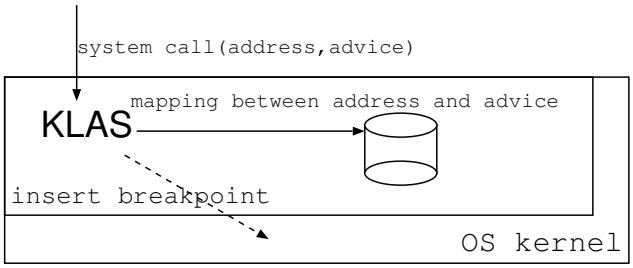

Figure 3: KLAS in the kernel space

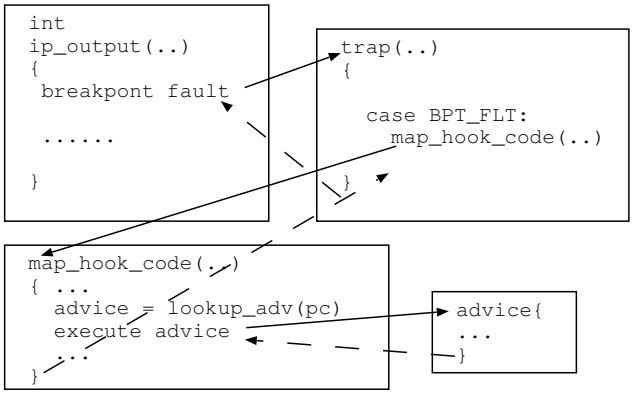

Figure 4: Execution of Advice

KLAS (Figure 3). This replacement is done while the OS kernel is running. Since the length of the breakpoint instruction of the x86 architecture is one byte, any machine instruction can be replaced with the breakpoint instruction. If the aspect is unwoven, the original machine instruction is substituted for the breakpoint-trap instruction. Note that the jmp instruction cannot be used as the hook code since the length of that instruction is three bytes. If an one-byte instruction located at the end of an basic block is replaced with the jmp instruction, the first instruction of the adjacent basic block is overwritten by the jmp instruction. This may cause system hang-up. However, according to our experiment, using the jmp instruction for the hook is about 25 times faster than using the breakpoint-trap instruction. We are planning extend KLAS to use the jmp instruction if the joinpoint is in the basic block.

When the thread of control reaches the breakpoint instruction substituted by KLAS, a breakpoint trap occurs (Figure 4). Then the trap handler executes the map_hook_code function, which we implemented. This function looks up the advice body corresponding to that breakpoint instruction, that is, the joinpoint and then executes that advice body. Finally, this function executes the original instruction replaced with the breakpoint instruction. KLAS execute this in the same way as DDB, the kernel debugger of FreeBSD, does.

4. RELATED WORK

4.1 Kernel Profilers

Ktr[8] is a kernel profiler included in BSD/OS and FreeBSD. This system is activated if a special compile option is given when the kernel is compiled. The users can manually insert logging code in the kernel source files before compilation. The logging code must be written by using CTRx macros. Since each logging code is individually activated during runtime according to the data structure called ktr_mask, the users can turn on and off the logging code by accessing ktr_mask. Unlike KLAS, Ktr does not allow the users at runtime to change the locations where log messages are printed. If the users insert logging code at a large number of locations, they can selectively activate only a few of them and change which ones are activated during runtime. However, this approach implies non-negligible overhead.

LKST[1, 2] is a kernel profiler for Linux. This system allows the users to record current time or execute any given code at a fixed set of locations in the kernel code. A problem of LKST is that the users cannot specify the locations where log messages are printed. They must select from the locations predetermined by LKST. This feature makes it difficult to investigate the behavior of the kernel in a fine-grained way.

KernInst[13] and GILK[9] can dynamically transform the binary code of the OS kernel. However, the users must specify which machine instructions are replaced with another code fragment. They cannot specify the replaced code with the source-level abstraction, such as a function call and a member access.

4.2 Aspect-oriented Solutions

AspectC++[12] is an aspect oriented system for the C++ language. Although it satisfies most of our requirements, it is a static aspect-oriented system. If the users change profiling code, that is, aspect code, the OS kernel must be recompiled and rebooted.

μDyner[11] is a dynamic aspect-oriented system for the C language, but the runtime overhead is not negligible. It inserts special *hook* code at the shadow of all the join-points at compile time. Some of the inserted hooks are activated by the pointcut description given at runtime and then they invoke an advice body when the thread of control reaches those hooks. That is, μDyner inserts the hook code at compile time at all the places in which the users may potentially want to measure the execution time. Since the number of the places in which the hook code must be inserted is usually large, the overhead due to the hook code is not negligible.

TinyC^2[7] is another dynamic aspect oriented system for the C language. Unlike μDyner, TinyC^2 can directly insert and remove the hook code in/from the compiled binary during runtime. This capability is provided by Dyninst[4], which is the backend system of TinyC^2. Since the hook code is inserted at only the places selected at runtime according to the given pointcut description, the overhead due to the hook code is minimized. However, TinyC^2 provides only a limited kind of execution points as joinpoints. For example, function calls are joinpoints but member accesses are not since the compiled binary of a C program does not include the information about which machine instruction corresponds to member accesses. The users must explicitly specify which machine instruction they want to pick out by pointcut description.

TOSKANA[3] is a dynamic aspect oriented system for OS-kernels. It runs on the NetBSD operating system. It uses the same approach as KLAS for loading advice. Since the hook is implemented with a branch instruction, the execution time of advice is faster than KLAS's. However, since it does not use a modified compiler, it cannot pick out a member access as a joinpoint.

AspectC is an aspect oriented system for the C language. Research with AspectC showed that AOP is useful for implementing the cache mechanism in the kernel[6, 5] . Caching in the kernel is a concern crosscutting across a memory management module and a disk management module. Since these two modules are in different layers, implementing this concern without AOP is more difficult than other concerns cutting across multiple modules at the same layer. AspectC weaves an aspect by source-to-source translation at compile time. It does not support dynamic weaving. Also it does not provide member accesses as joinpoints.

PROSE[10] is an early dynamic AOP system for Java. It uses JVMDI (Java Virtual Machine Debugger Interface) to implement dynamic weaving of aspects. It sets breakpoints at the joinpoints specified by pointcuts. If the thread of control reaches one of those breakpoints, the JVM (Java Virtual Machine) transfers the control to the PROSE system so that PROSE will execute advice code associated with the joinpoint. This idea is the same as ours but KLAS is a dynamic AOP system for the OS kernel written in C. The cost of handling breakpoint traps in the OS kernel is relatively smaller than in the JVM.

5. CONCLUSION

Investigating a network bottleneck in the OS kernel needs a sophisticated kernel profiler that enables measuring execution time of a fine-grained code section. Since the OS kernel consists of a large number of layered modules, aspect orientation is a significant paradigm for designing such a kernel profiler. Furthermore, an aspect-oriented kernel profiler should be able to dynamically weave an aspect for avoiding kernel rebooting, which seriously decreases the efficiency of the investigation of performance bottleneck.

In this paper, we proposed an aspect oriented system named KLAS. This system provides fine-grained joinpoints, including member accesses to structures, so that the users can investigate details of the behavior of the OS kernel. KLAS extends the symbol information included in the compiled binary. KLAS collects symbol information, such as the file name and the line number of member-access expressions, at compile time and it makes the collected information available for the runtime weaver of KLAS. This is because normal C compilers produce a relatively smaller amount of symbol information than Java compilers. We are currently still implementing KLAS. The runtime weaver and the enhanced the GNU C compiler have been implemented but a language processor for compiling an aspect written in XML are still being implemented.

6. REFERENCES

[1] `http://lkst.sourceforge.net/`.

[2] `http://oss.hitachi.co.jp/sdl/english/lkst.html`.

[3] `http://www.betriebssysteme.org/Aktivitaeten/Treffen/2004-Dresden/Programm/Folien/Engel/AOSTA-Slides.pdf`.

[4] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[5] 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.

[6] 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.

[7] G. T. Leavens and C. C. (eds.). Foal 9,003 proceedings - foundations of aspect-oriented langauges workshop at aosd 2003.

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

[9] 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.

[10] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM Press, 2002.

[11] 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.

[12] 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.

[13] 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.