A Dynamic Aspect Oriented Kernel Profiler for Ease of Use

動的アスペクト指向に基づいた使いやすいカーネルプロファイラ

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

by

Yoshisato YANAGISAWA

柳澤 佳里

03M3731-1

January 2005

A Master's Thesis Submitted to
Department of Mathematical and Computing Sciences
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

In Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Supervisor: Shigeru Chiba

Copyright © 2005 by Yoshisato YANAGISAWA. All Rights Reserverd.

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, join points 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 5.2.1 operating system with the GNU C compiler. Experiment shows that our modified C compiler spent much time compared with normal C compiler. Experiment also represent that execution overhead of KLAS is much smaller when a number of aspects is small. However, experiment also showed that overhead becomes much larger when KLAS runs with a quite large number of aspects.

Acknowledgments

This thesis was supervised by Shigeru Chiba. I would like to express my respect for him.

I greatly thank Kenichi Kourai. He spent a large amount of time for me and gave me a lot of suggestions.

Moreover, I thank my colleagues of the Chiba group at the Tokyo Institute of Technology. In particular, discussions with Yoshiki Sato, Muga Nishizawa and Masahiro Matsunuma were very interesting.

Contents

1 Introduction **1**

2 Profiler for OS-kernel **6**
2.1 Requirements . . . 6
2.1.1 Changeability of Code and Points without Rebooting 7
2.1.2 Supporting the C language . . . 8
2.1.3 Fine-grained Profiling . . . 8
2.1.4 Minimized Prove Effects . . . 10
2.2 Existing Tools . . . 10
2.2.1 Aspect-oriented Programming . . . 11
2.2.2 AspectC++ . . . 13
2.2.3 μ-Dyner . . . 13
2.2.4 TinyC^2 . . . 14

3 KLAS: Kernel Level Aspect-oriented System **15**
3.1 Overview of the KLAS system . . . 15
3.2 Implementation . . . 16
3.2.1 Mechanism to Insert Hook . . . 20
3.2.2 Mechanism to Collect Symbol Information . . . 26
3.2.3 Mechanism to Find Address . . . 29
3.2.4 Parsing Aspects and Weaving Mechanism . . . 41

4 Experiment **44**
4.1 Accuracy of Positions of Hooks . . . 44
4.2 Simple Performance Measurement . . . 45
4.2.1 Performance Measurement of Kernel Compile . . . 45
4.2.2 Performance Measurement of Execution . . . 45
4.3 Performance Measurement of Network Codes . . . 48

5 Related Work 50
5.1 Kernel Profilers . . . 50
5.1.1 Ktr . . . 50
5.1.2 LKST . . . 50
5.1.3 KernInst and GILK . . . 51
5.2 Other Aspect-oriented Solutions . . . 51
5.2.1 AspectC . . . 51
5.2.2 PROSE . . . 52
5.2.3 Wool . . . 52

6 Concluding Remarks 53

List of Figures

1.1 Example: sys/netinet/tcp_output.c 2

2.1 The way finding performance bottlenecks 7
2.2 fileops structure(sys/file.h of FreeBSD) 9

3.1 An Overview of KLAS 16
3.2 Aspect Definition in KLAS 17
3.3 Implementation of KLAS 19
3.4 KLAS in the kernel space 20
3.5 Execution of Advice 20
3.6 Difference between Usual Debugger and DDB 21
3.7 Symbol tables in DDB 23
3.8 Call Graph to Find an Address 24
3.9 Execution of Breakpoint-trap(BPT) 26
3.10 The Way to Map ID and the Name 28
3.11 Organization of Debugging Information 30
3.12 Decode unsigned LEB128 number 32
3.13 Decode signed LEB128 number 32
3.14 Parsing a Special Opcode 39
3.15 Action of KLAS Userland Half 42
3.16 Action to Resolve Addresses 43

4.1 Elapsed Time by Increasing a Number of Aspects 46
4.2 Elapsed Time by Increasing a Number of Aspects (Log-scale) 47
4.3 Elapsed Time with a Smaller Number of Aspects 47
4.4 Elapsed Time with a Smaller Number of Aspects (Log-scale) 48

List of Tables

2.1 Several pointcut designators of AspectJ 12
2.2 Advice of AspectJ 13

3.1 Attribute format encodings 35
3.2 Terms of Line Number Information Format 36
3.3 Initial State of State Machine 37

4.1 Elapsed Time for Compiling OS Kernel 46
4.2 Throughput Change of Inserting Advice 49

Chapter 1

Introduction

During the history of operating systems (OSes), 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, block allocation algorithms of filesystems, implementation of network stack, lock mechanism, and so on. For example, both FreeBSD and Linux recently introduced new implementation of their process schedulers.

Investigating a performance bottleneck is one of the important first step for improving the performance of OS kernels. Investigating a performance bottleneck let developers find the reason of performance degradation. It helps developers to decide the way how to minimize such bottlenecks or improve performance of these codes. If codes including a performance bottleneck is not required to process realtime it can be delayed until CPU becomes idle to improve peek time performance. It is similar solution to filesystem delayed-write or network delayed-ack. Improving such a bottleneck might be great help for performance improvement since there are rule of thumb that more than 80% of time is elapsed in less than 20% code snippets.

To investigate a performance bottleneck, using a performance profiling tool for OS kernels is mandatory. A performance profiling tool is a tool that measure elapsed time between certain points in the OS kernel. In fact, we are studying a performance bottleneck for network processing, that is, inappropriate behavior of the network module of the FreeBSD operating system when multiple process are executing network I/O simultaneously. Since there are many modules and functions associated with network I/O, a performance profiling tool should be used. Developers would first measure the execution time of a large code section and then they would gradually

```
int
tcp_output(struct tcpcb *tp)
{
    ...
    struct mbuf *m;
    ...
    m->m_len = hdrlen;
    ...
}
```

Figure 1.1: Example: sys/netinet/tcp_output.c

narrow the range of that code section to find a performance bottleneck. Since rebooting need much time and cause loss of memory, performance profiling tool, which can change the points to measure time is needed.

Motivating Problem

Existing performance profiling tools for OS kernels are not powerful enough to investigate a performance bottleneck in detail. There are two kinds of performance profiling tools: tools which can get logs at fixed point and tools which can transform codes to insert logs at any points. However, neither is powerful enough. To investigate a performance bottleneck of network I/O, measurement of elapsed time between member accesses of mbuf [1] structure is required. That is because this structure is often used inside network code.

Figure 1.1 is an example to represent a weakness of existing tools. Since whole network access should be traced to investigate a performance bottleneck, developers should measure elapsed time of each mbuf member access. Since a location of each member access is removed by a compiler, existing performance profiling tools, which only use information of running binary can not designate the program counter to get logs. All member accesses are changed into ld or st instruction and can not be distinguished.

Performance profilers which can get logs at fixed point is not powerful enough. Some of these tools can get logs at entry-point and exit-point of a function and the others can get log at occurrence of some specified events. Since the point to get a log is restricted, these tools can not trace network I/O codes, especially mbuf member accesses. Former one doesn't

[1] memory buffer used in network stack codes of the FreeBSD operating system

support these kinds of points because the positions of them are disappeared at compile time and can not investigate these positions at runtime. Latter one doesn't support these kinds of points either because the positions of them are restricted already by developers of the tool. Even if a developer increases a number of these points, measurement effects become so large that a developer can not get logs in detail.

Dynamic code transformation is not powerful enough to trace network I/O. A developer can insert log codes into anywhere he or she wants by using dynamic code transformation. Then he or she can measure elapsed time between certain points. However, since a compiler converts a member access of a structure into a ld or st instruction and the program counter of the instruction is not stored anywhere, a developer can not point out such structure access to get a log.

Solution

To solve the problem mentioned above, this thesis proposes a new dynamic code insertion mechanism for enabling insertion of code snippets into a point of a member access. With the proposed mechanism, developers can get logs at two kinds of member accesses: calling a function pointer and accessing a data. Since accessing a data of a member is often used inside network I/O codes of an OS kernel, developers can trace the network I/O code by using this feature.

The dynamic code insertion mechanism proposed by this thesis is called "Source-based binary-level dynamic weaving" and the name of a kernel profiler presented by this thesis is *KLAS*. It means that KLAS uses a source code level information of a member access to pick out the execution point to insert code snippets dynamically. Since most of source level information disappears at compile time, existing dynamic code insertion mechanism can not pick out these points.

To get source-level view, KLAS uses an extra symbol information. An extra symbol information is generated by a compiler modified for KLAS. It is used by KLAS at the time it inserts code snippets into a running OS kernel. Usual symbol information has an information of a name of a function and a mapping of a line and a program counter. An extra symbol information this thesis proposes has an information of a line where a member of a structure is accessed. KLAS use both symbol informations to pick out the point to insert code snippets.

An extra symbol information contains a line number and a file name of a

member access. To pick out a program counter of a member access, KLAS uses an extra symbol information to get a line number and a file name. Then KLAS use an usual symbol information to get a program counter with them.

KLAS uses breakpoint-trap instructions to insert code snippets. To insert code snippets into a OS kernel, KLAS replaces machine instructions on an OS kernel with breakpoint-trap instructions so that code snippets can be executed at the addresses of those instructions. The place where breakpoint-trap instructions inserted and the program counter associated with it is remembered by KLAS to execute a required code snippet at the time breakpoint-trap occurred.

We developed KLAS on the FreeBSD operating system[17] version 5.2.1 and the GNU C Compiler version 3.3.4. The FreeBSD operating is an open source operating system derived from UC Berkeley's 4.4BSD operating system[16].

The Structure of This Thesis

From the next chapter, we present background, system overview and implementation issues of KLAS. The structure of the rest of this thesis is as follows:

Chapter 2: Profiler for OS-kernel

At first, we describe requirements for kernel profilers for finding out performance bottlenecks to clear our motivating problems. Then we explain problems of the existing tools.

Chapter 3: KLAS: Kernel Level Aspect-oriented System

To address the motivating problem, we present our new implementation technique for dynamic AOP. It also shows an overview of the KLAS system.

Chapter 4: Experiment

At first, we explained possibility and limitation of a hook insertion of the KLAS system. Next, we measured the elapsed time for compiling OS kernel to examine performance degradation with our gcc modification. Then we measured elapsed time of executing the code snippets where KLAS weaved aspects to examine performance degradation by KLAS.

Chapter 5: Related Work

At first We compared KLAS and kernel profilers and explained advantages of KLAS. Then we compared KLAS and other AOP systems which are not discussed in chapter 2.

Chapter 6: Concluding Remarks

We conclude this thesis in chapter 6. Moreover, we present future work.

Chapter 2

Profiler for OS-kernel

Detecting a performance bottleneck in an operating system (OS) kernel is an important topic of operating system study. In fact, we are studying a performance bottleneck for network processing, that is, inappropriate behavior of the network module of the FreeBSD operating system when multiple processes are executing network I/O simultaneously.

To investigate such 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 investigating kernel performance. One profiler allows us to measure elapsed time between any given two function calls. 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 and problems of the existing tools and techniques.

2.1 Requirements

There are four requirements for a kernel profiler. Without fulfilling these requirements, efficiency of profiling should become too bad. Requirements are as follows.

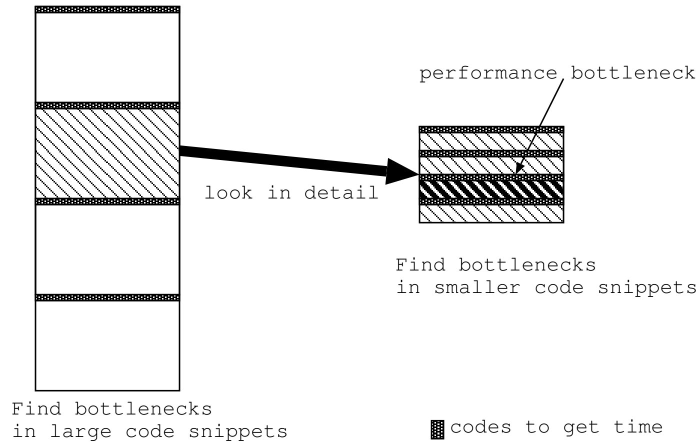

Figure 2.1: The way finding performance bottlenecks

2.1.1 Changeability of Code and Points without Rebooting

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 users also must be able to give code snippet for measuring the elapsed time at runtime and change them 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 (Figure 2.1). This might be the most efficient way to find a performance bottleneck. That is because finding a performance bottleneck in short code section tend to make the user miss the cause of a performance bottleneck. Since rebooting the whole kernel is a time consuming task, frequent rebooting significantly decreases our productivity. In fact, elapsed time between before and after rebooting the FreeBSD operating system is about one minute and the user should wait this long time every time they change the execution points. 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. Then the users should wait until the behavior appears again.

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. The code snippet for measuring the elapsed time should be given and changed without rebooting the kernel since rebooting the kernel needs much time and may vanish some behavior the user want to investigate. Also, the users may want to print a log message, for example, to record the value of an interesting variable.

2.1.2 Supporting the C language

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 and NetBSD, are written in C language.

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. With these features, execution binary of C language can be run fast instead of being ease of maintenance, and can be small not to waste space and easy to use with embedded devices.

2.1.3 Fine-grained Profiling

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. Function pointer is often used in operating system(OS) kernel and is sometimes included in a structure as a member. A number of execution points that we are interested in for performance profiling are function calls through function pointers.

Modern OS kernels uses 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. It means that a file descriptor which passed with `read` or `write` system call is just integer value and is not be able to distinguish file descriptor is made by `socket`

```
typedef int fo_rdwr_t(struct file *fp, struct uio *uio,
                    struct ucred *active_cred, int flags,
                    struct thread *td);
#define FOF_OFFSET      1       /* Use the offset in uio argument */
typedef int fo_ioctl_t(struct file *fp, u_long com, void *data,
                    struct ucred *active_cred, struct thread *td);
typedef int fo_poll_t(struct file *fp, int events,
                    struct ucred *active_cred, struct thread *td);
typedef int fo_kqfilter_t(struct file *fp, struct knote *kn);
typedef int fo_stat_t(struct file *fp, struct stat *sb,
                    struct ucred *active_cred, struct thread *td);
typedef int fo_close_t(struct file *fp, struct thread *td);
typedef int fo_flags_t;

struct fileops {
        fo_rdwr_t       *fo_read;
        fo_rdwr_t       *fo_write;
        fo_ioctl_t      *fo_ioctl;
        fo_poll_t       *fo_poll;
        fo_kqfilter_t   *fo_kqfilter;
        fo_stat_t       *fo_stat;
        fo_close_t      *fo_close;
        fo_flags_t      fo_flags;       /* DFLAG_* below */
};
```

Figure 2.2: fileops structure(sys/file.h of FreeBSD)

system call or open system call. Read or Write routine of socket I/O or File I/O, however, is associated with file descriptor for the member of a fileops structure (Figure 2.2) at the time the users call open or socket system call and easily dispatched when the user calls read or write system call. Here is another example. The VFS (Virtual File System) uses the same technique for dispatching to a function appropriate to each type of file system. To use many kind of file system in an Unix operating system, it has a mechanism for hiding real file system and making them behave as an Unix own file system. This mechanism is called VFS and realizes the behavior which suit for each 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. To realize changeability of behavior to free a mbuf cluster, a set of large memory to use large data in network I/O, a function for freeing mbuf cluster is a function pointer and is the member of mbuf structure.

2.1.4 Minimized Prove Effects

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. That is because the sum of elapsed time for measuring would cover the elapsed time where the performance is not good if the overheads of measurement is large. Moreover, if the sum of elapsed time for measurement is larger than the elapsed time of executing the code snippet whose performance is not good, the users can not find these code snippet any more.

Once necessary data is 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. To realize this, measurement code must be easily invalidated sooner after interesting data is gotten. Then a method inserting a tab for executing a time measurement code is not a good way to realize code changeability. That is because the code snippet of such a tab consume a time everywhere a time measurement code is executed and will cover a elapsed time of a bottleneck.

the size of a measurement code is another problem. If code size of OS kernel is too large, memory amount which application can use becomes small and a number of page in and out will becomes many. the size of a measurement code would not be so large that application cause page in and out frequently. If a tab, however, is inserted anywhere a joinpoint exists, the sum of every code would become too large to run applications and kernel efficiently even if a tab is small. That is because a number of joinpoint would be too large if all functions and member accesses to structures become joinpoints.

2.2 Existing Tools

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 after recompiling source codes. Although the most promising approach is to use the idea of aspect-

oriented programming (AOP), the existing AOP-based tools do not satisfy our requirements.

2.2.1 Aspect-oriented Programming

Aspect-Oriented Programming(AOP) has been proposed as technology for improving separation of concerns in software. Although object-oriented programming(OOP) is the technology that can fundamentally modularize software systems, it is not sufficient technology enough to separate concerns that are scattered throughout modules. We call such concerns *cross cutting concerns*. They decrease maintainability and understandability of software systems. The users must understand and change each modules in software even if they change and maintain the crosscutting implementation because the crosscutting implementation is scattered throughout modules.

Logging is good example for understanding a strong point of AOP. The users would scatter logging code to programs in developing phase to find bugs and would remove such logging code after development finished. When they remove such codes for logging, they should change the programs they inserted codes for logging at the time of development to remove the codes. Such operation, however, is prone to make other software bugs because they change these codes by hands, in another words, human can easily make easy mistakes. By using AOP instead of hand-coding, the users can easily insert and remove codes without causing a care-less mistakes. That is because AOP can pull the widespread crosscutting concern into a single module, and inserting and removing the codes is done by a unit of such a module. These modules are termed *aspects* in AOP. AOP builds on several technologies, such as procedural programming and OOP, which have already made significant improvements in software modularity.

AspectJ

Before representing each AOP system, let us introduce AspectJ [13, 8, 11]. AspectJ is a famous AOP system and syntax of many AOP systems are similar to AspectJ. In AspectJ, there are three important elements: a *joinpoint*, a *pointcut* and an *advice*. A pair of some *pointcut*s and *advice*s is called an *aspect*. It is used passed to an AspectJ and it will insert a code given in an *advice* body to the point where a pointcut point out. We call this operation *weave* and the contrary operation, which removes the codes from the program codes, is called *unweave*.

designator	meaning
set(*Signature*)	select points writing a field of *Signature*
get(*Signature*)	select points reading a field of *Signature*
call(it Signature)	select points calling functions of *Signature*
execution(*Signature*)	select points executing functions of *Signature*
cflow(*Pointcut*)	all joinpoints that occur between the entry and exit of each joinpoint specified by *Pointcut*

Table 2.1: Several pointcut designators of AspectJ

Joinpoint A joinpoint is the execution point which is defined in program execution flow and is the point which is calling a function, executing a function or accessing a variable. We means that the point calling a function, for example, is a point which represent the points at the beginning and ending of a function call. Life time of this joinpoint is from beginning and ending of a function call, yet the point represented by joinpoint is only an instance of beginning and ending a function call.

Pointcut *Pointcut* is used to pick out the specific points from *joinpoints* in the program cflow. The following pointcut designator, for example, picks out joinpoints that is a function call of the signature void Point.setX(int) in the program flow.

call(void Point.setX(int))

The call is one of pointcut designators, identifies each join points that are calls of the specified functions. In table 2.1, we listed several pointcut designators that AspectJ is provided. A pointcut can be build out other pointcuts with and (&&), or(||) and not(). Following example picks out joinpoints that is a function of the signature void Point.setX(int) or void Point.setY(int).

call(void Point.setX(int)) || call(void Point.setY(int))

Advice An advice defines the code which is executed at a joinpoint and a pointcut to execute the code. AspectJ has *before*, *after* and *around* advice and these are described in table 2.2.

advice	meaning
before	execute the code before a pointcut
after	execute the code after a pointcut
around	execute the code instead of executing a code snippet at a pointcut

Table 2.2: Advice of AspectJ

2.2.2 AspectC++

AspectC++[23, 25] is an aspect oriented system for the C++ language. Syntax of this system is very similar to AspectJ. It supports call and execution pointcuts and also supports set and get pointcuts. Difference of grammar between AspectJ and AspectC++ is follows: a wildcard operator is '%' instead of '*' in AspectC, an operator which joins a class name and a method or field name is '::' instead of '.', and a pointcut designator is quoted by '"' to treat it a string though no quoting is used in AspectJ.

AspectC++ is using a source code translator PUMA[18] to realize aspect weaving. First source code for AspectC++ is scanned and parsed by PUMA. Then AspectC++ get syntax tree generated by PUMA and make manipulation commands to weave codes for AspectC++. Finally PUMA manipulation engine get manipulation commands and weave these codes into C++ source codes. Codes generated by PUMA is only written in C++ language and can be compiled by normal C++ compiler.

AspectC++ satisfies most of our requirements. It support a pointcut to a member function of a class and might be support a member field of a structure which is a function pointer. It also support C language because C++ language is expanded from C language and it only weaves codes to the points where pointcuts is designated. It is, however, a static aspect-oriented system and cannot weave codes at runtime. If the users change profiling code, that is, aspect code, the OS kernel must be recompiled and rebooted.

2.2.3 μ-Dyner

μ-Dyner[22] is a dynamic aspect-oriented system for the C language. This system is made to realize web cache prefetching mechanism by dynamic AOP. It means that the user write cache policy by an aspect and pass them to μ-Dyner to change cache policy. Caching is one of cross cutting concern and code snippets to realize caching is spread to whole source codes. By

using dynamic AOP, however, the user can give cache policy easily and easily change cache policy.

The runtime overhead is, however, 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, μ-Dyner inserts the hook code at compile time at all the places in which the users may potentially want to measure the execution time. These special codes are consist of **jmp** and **nop** instructions and these instructions are thought to be light weight. Yet a number of the places in which the hook code must be inserted is usually large for our purpose, profiling OS kernel, and the overhead due to the hook code is not negligible. Moreover, since a number of hook code is large, the size of hook codes is large as to consume much time at running OS kernel for page in and out.

2.2.4 TinyC^2

TinyC^2[14] 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[2], 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 join points. For example, function calls are join points 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.

Chapter 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. 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

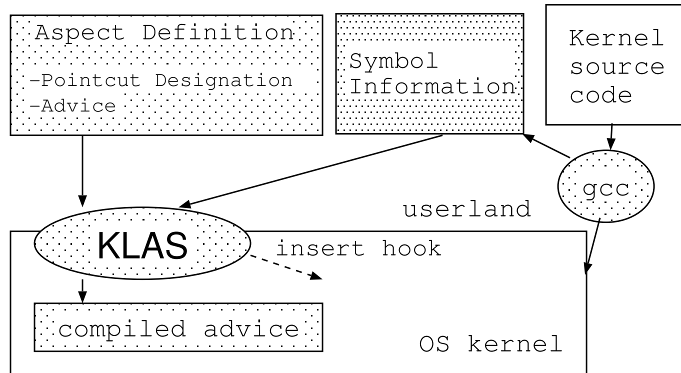

Figure 3.1: An Overview of KLAS

since we can easily measure the execution time of functions accessing the mbuf structure.

An aspect definition for KLAS is described in XML. KLAS supports only before advice. Figure 3.2 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 dynamically inserts the hook code only at the places corresponding to the join point 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

```
<aspect name="log_mbuf_clean">
  <pointcut>
    <field name="ext_free" structure="m_ext" />
  <pointcut>
  <advice>
    void* resolve_argument(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));
  </advice>
</aspect>
```

Figure 3.2: Aspect Definition in KLAS

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 3.3).

To use KLAS, the OS kernel must be compiled by our extended 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, 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 and GNU C compiler uses those id. numbers for distinguishing 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 join points 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 join point, it cannot insert the hook code exactly at the instruction corresponding to that join point. However, this limitation is not a serious problem for our application, which is investigating a performance bottleneck of the OS kernel.

KLAS uses 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

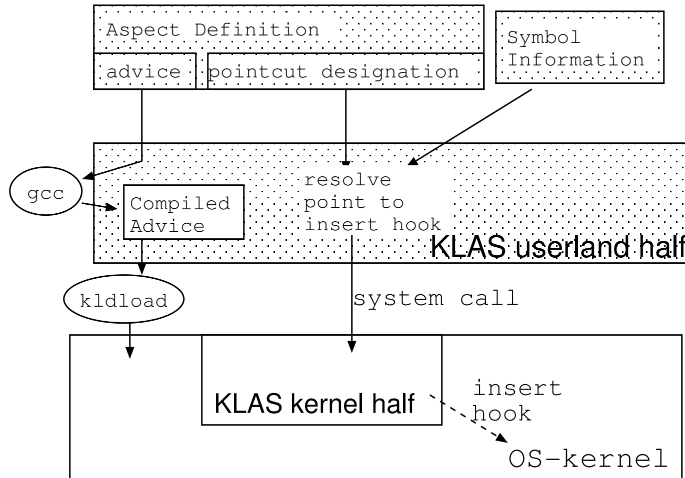

Figure 3.3: Implementation of KLAS

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 join point and replaces it with the breakpoint-trap instruction, which is the hook code of KLAS (Figure 3.4). 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.

When the thread of control reaches the breakpoint instruction substituted by KLAS, a breakpoint trap occurs (Figure 3.5). 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 join point and then executes that advice body. Finally, this

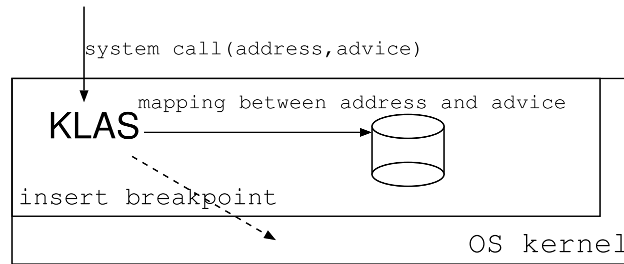

Figure 3.4: KLAS in the kernel space

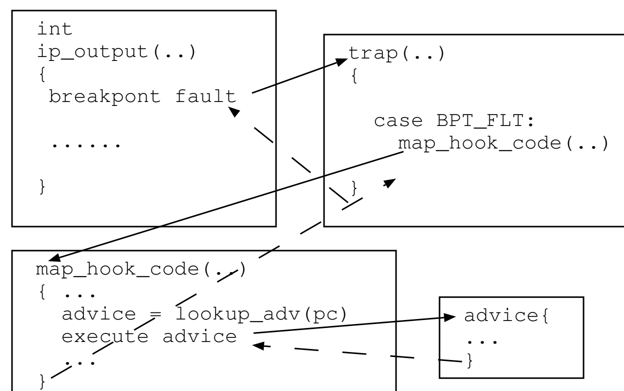

Figure 3.5: Execution of Advice

function executes the original instruction replaced with the breakpoint instruction.

Following subsection represent implementation of KLAS in detail. We are going to represent each module, one by one.

3.2.1 Mechanism to Insert Hook

Hook insertion mechanism is one of the most important mechanisms of KLAS. With this mechanism, KLAS can insert hooks into OS kernel without rebooting. This mechanism is using a thought of DDB, one of the FreeBSD kernel debugger. Before talking about a hook insertion mechanism of KLAS, let us talk about DDB.

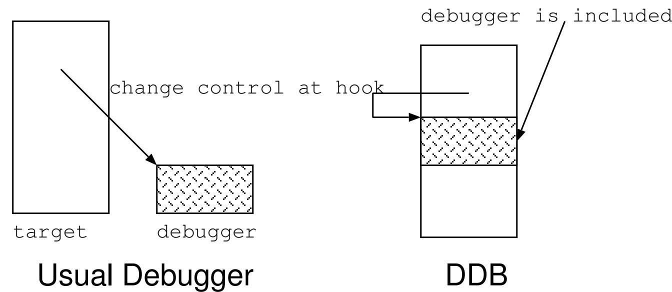

Figure 3.6: Difference between Usual Debugger and DDB

DDB

DDB is a famous kernel debugger in FreeBSD. It was developed on the Mach operating system, an operating system made by CMU computer research team, and ported to 386BSD. Instead of usual debugger like GDB, GNU debugger, DDB is included in OS kernel and runs on same CPU. Usual kernel debugger runs the other system and would not stop its action even if target system of debugging goes down. DDB, however, runs the same system with OS kernel and once the break point trap or some other trap, which should move control to a debugger, is occurred, the control is jumped into the debugger code snippet in the same program (Figure 3.6). By this mechanism, execution of DDB has nothing to do with execution speed of a target operating system.

We would like to explain design and implementation of DDB one by one. First, we will explain the action after trap and then explain the symbol table structures debugger has and then explain the way how to look up address from the symbol table. Finally, we explain the way to insert breakpoint-trap by DDB.

Action after Trap Occurred Codes of action after trap is including both machine dependent codes and machine independent codes. So, if we explain the action, we should explain each trap treatment mechanism of each architecture. Yet We would like to explain only the implementation of Intel 80x86 architecture since the architecture FreeBSD is supporting mainly and whose number of users is the most in FreeBSD is Intel 80x86 architecture.

At the time breakpoint-trap or some other trap occurred, alltraps function, which is in sys/i386/i386/exception.s file is called. This is a general

trap management routine of FreeBSD and whole trap is registered to interrupt vector in this file(exception.s). These traps are associated with a trap management function written in the C language, named trap. Before calling this function, alltrap push some registers onto stack.

Trap function is in sys/i386/i386/trap.c file and it manages whole traps including a breakpoint-trap and a page fault trap and so on. Trap function gets trapframe structure from alltrap function and trapframe structure has a information about a kind of a trap which causes trap function called and values of registers at the time trap occurred and so on. In trap function, proper routine is chosen by the kind of a trap; for example, SIGPFE is set and returned for the arithmetic trap, and page fault management routine is called for page fault. If breakpoint-trap occurs, trap function first distinguish it is called at userland or kernel. If it occurs in userland, SIGTRAP is set and returned. If it occurs in kernel, kdb_trap function, general kernel debugger routine, is called.

Kdb_trap function is general debugger function and is in sys/kern/subr_kdb.c. This function is machine independent function and this will be called by trap management routine of each architecture. In FreeBSD, gdb, GNU debugger, and DDB, debugger which comes from the Mach operating system, can be chosen. By way of parenthesis, functions in this file is like an interface of Java and they accept difference between gdb and DDB. The member of the structure kdb_dbbe holds a pointer to a debugger entrance function and this is one of polymorphism like mechanism. At this point, db_trap is called by kdb_trap function if DDB is chosen as a kernel debugger.

In db_trap function, lists of addresses which stored breakpoint-trap and watchpoint-trap, watchpoint has not been implemented yet though, is scanned. Then print proper message with a kind of a trap and finally, command loop, which receive command from user in command line, will be executed. After command loop, kernel will run from the point where trap occurred.

Structure of db_symtabs and Its Members DDB has symbol tables inside it to find a proper address of a function from a name of a function and to find a proper name of a function from an address. DDB resolves addresses or names of functions by using tree structures: db_symtabs structure, a symbol table structure and Elf_Sym structure (Figure 3.7).

Members of db_symtabs structure is made for each symbol tables; an elf symbol table, a symbol table for symbols in compiled kernel, and an kld symbol table, a symbol table for symbols which are loaded by kldload command dynamically. Each member point out the symbol table for it and

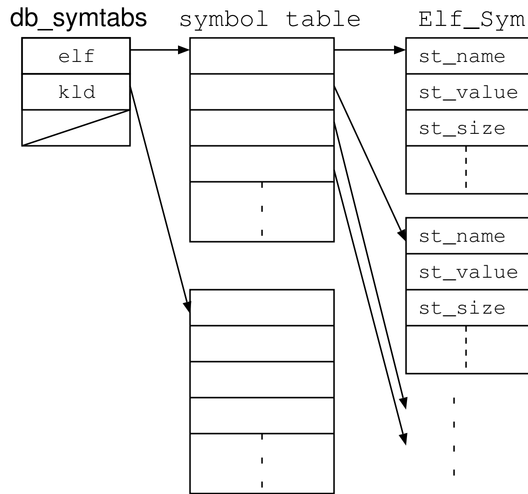

Figure 3.7: Symbol tables in DDB

a member of symbol table point out Elf_Sym structure.

Elf_Sym structure contains a name and an address of each function: st_name describes the name of the function and st_value describes the address of beginning the function. This structure is in ELF binary and is set by db_add_symbol_table function at the beginning of ddb, when db_init function is called.

The Way to Look up the Function Address Address of beginning of the function is needed to insert breakpoint-trap by function name or to call the function inside DDB. This paragraph describes the way to find address of the function from symbol tables described before.

To find the address of the function, db_value_of_name function, which is in sys/ddb/db_sym.c, is called (Figure 3.8). Arguments of this function are the name of the function and the pointer to db_expr_t structure to return an address. The name of the function is passed to db_lookup function. This function passes it to X_db_lookup function with a pointer to a structure of each symbol table. Then X_db_lookup function traces symbol table to find Elf_Sym structure whose st_name field is match the name of the function given as an argument. X_db_lookup returns the pointer to the Elf_Sym structure and then db_lookup returns it to db_value_of_name function. Then db_value_of_name function pass this pointer value, the pointer to Elf_Sym structure whose

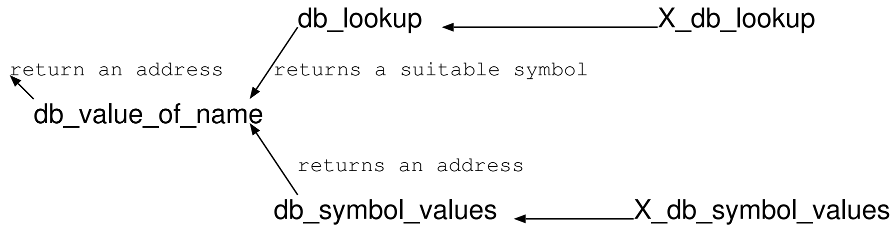

Figure 3.8: Call Graph to Find an Address

member st_name is matched to the function name, to db_symbol_values function. Db_symbol_values function calls X_db_symbol_value with the argument: the pointer and the name of the function. X_db_symbol_values distinguish whether the passed pointer is from elf symbol table or kld symbol table. If the pointer is from elf symbol table, the value of st_value, the member of Elf_Sym structure, is returned. Otherwise, address is resolved with linker_ddb_symbol_values function, which find a symbol table of dynamic loaded functions. Then db_symbol_values function return the returned value of X_db_symbol_values function. Db_value_of_name function returns the value and the caller of this function successfully get the address of the function.

The Way to Insert Breakpoint-trap Breakpoint-trap is used to change control from operating system to debugger at any point the user want. This paragraph describes the way to insert breakpoint-trap after the user gives *break* command to DDB.

After *break* command is given by the user, db_break_cmd function will be called. Db_break_cmd calls db_set_breakpoint function. Db_set_breakpoint function first calls db_find_breakpoint function to distinguish whether a breakpoint-trap already set at the address or not. If a breakpoint-trap has already set, it will print "Already set" and return. After this, db_set_breakpint function will call db_breakpoint_alloc function to get memory to store the address to insert breakpoint-trap and the address map of it. Then db_set_breakpoint function set the address and the address map to db_breakpoint_t type structure allocated by db_breakpoint_alloc function to save breakpoint-trap information and add it to linked list of breakpoint-traps whose name is db_breakpoint_list.

At the time kernel restarted, *continue* command is given for example, db_set_breakpoints function is called. This function trace db_breakpoint_list and insert breakpoint trap into the point where the member of db_breakpoint_t type structure pointing. To insert breakpoint-trap, db_set_breakpoints func-

tion first calls db_map_current to set address map and then it use BKPT_WRITE macro. This macro substitute the instruction of the address and breakpoint-trap and store the instruction into the member of db_breakpoint_t type structure. db_clear_breakpoints will called before entering DDB command loop and breakpoint-traps are removed by this function.

KLAS Kernel Half Implementation

KLAS kernel half use breakpoint-trap to execute advice body at the point the user ordered. The way KLAS kernel half insert breakpoint-trap is based on DDB breakpoint-trap insertion mechanism mentioned before. KLAS kernel half receives the point and the name of the function advice body is included in to insert hook from the user by a system call. KLAS kernel half register the mapping of the address and the name of the function. Then KLAS kernel half insert a breakpoint-trap into OS kernel. Following shows the detailed implementation one by one.

Table mapping address and advice KLAS kernel half has a table which maps address of breakpoint-trap and the name of advice body function. KLAS kernel half use linked list to realize this table. The member of this table is tha pair of the address and the name. When breakpoint occurs, linked list is traced one by one to find the member which contains the address of breakpoint-trap.

System Call Management Routine We implemented a system call to insert and to remove the breakpoint-trap and the mapping of an address of breakpoint-trap and advice code. This system call manages the table mapping address and advice in backend. To insert the breakpoint-trap and the mapping, the user call this system call with two argument: an address to insert hook and a name of advice. The function attached to the system call first checks the address is registered or not. If registered, system call returns error value. Otherwise, the function add these arguments to the table explained before, and then call db_clear_breakpoints function to clear whole breakpoint-trap. Next, this function calls db_breakpoint_alloc function, one of the function managing breakpoint-trap in DDB, to add the address to db_breakpoint_list. Then this function calls db_set_breakpoints function to insert breakpoint-trap into OS kernel.

To remove the hook from OS kernel, the user calls system call for this purpose. The function attached to the system call first checks the address passed by the user is contained by the member of the list containing the

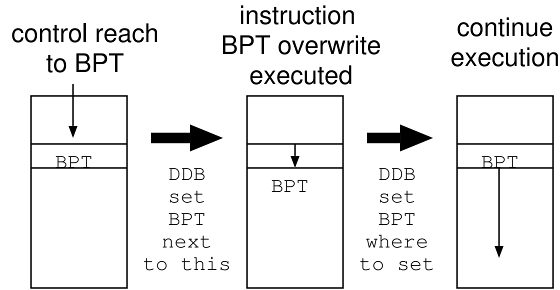

Figure 3.9: Execution of Breakpoint-trap(BPT)

mapping of address and advice or not. If not, it returns error. Otherwise, it calls db_clear_breakpoints to remove whole breakpoint trap. It remove the member from the list and db_breakpoint_list by using db_delete_breakpoint function. Then it calls db_set_breakpoints function to reflect the change.

Interface to DDB KLAS kernel half often use DDB functions to insert and remove a breakpoint-trap. The functions using DDB functions are gathered in one file to separate other modules and this one. One of function of this file is called when a breakpoint-trap which is set by KLAS kernel half occurs. We modified db_trap function to execute this function at the time breakpoint trap occurs. In this function, we set **epc**, **ebp** and **esp** registers to be able to use in advice codes.

KLAS kernel half calls db_continue_cmd function after executing advice code. To execute breakpoint-trap collectly, DDB insert breakpoint trap next to the point the breakpoint was inserted and remove the breakpoint-trap where it was to execute instructions at breakpoint-trap (Figure 3.9). This is done by db_continue_cmd function. When control reached to next instruction, breakpoint-trap occurs and DDB removes the breakpoint-trap there and insert breakpoint-trap previous of this point, the point where breakpoint should be inserted.

3.2.2 Mechanism to Collect Symbol Information

The unique mechanism KLAS has is using source-level view for dynamic weaving aspects to OS kernel. To realize this mechanism, we alternate GNU C compiler to collect symbol information. In following subsection, first we explain the parsing and analyzing meaning of structures in GNU C compiler

which we alternated to realize member access as joinpoint. Then we explain our mechanism to collect symbol information.

GNU C Compiler

GNU C compiler is implemented with kind of polymorphism to realize parsing of meaning in visitor pattern like mechanism. All elements are type of tree structure but tree structure is like an interface or an abstract class of Java. Tree structure just supply the uniformed size of space and the type itself has no members or meanings. Specific macros are prepared to use tree structure in source code. Each macro first cast tree type to other types needed at the point and then access its member. GNU C compiler changes symbols into id. numbers at parsing time and store them tree structure. It use only id. number instead of symbol name inside the code of compiler.

xref_tag C program is first parsed codes according to yacc or bison rule written in c-parse.in file, and then passed to xref_tag function in c-decl.c file. The arguments passed to xref_tag function is id. number and symbol information of the structure. We can get IDENTIFIER_POINTER macro to get the name of the structure.

build_component_ref At the time compiler parses a member of a structure, build_component_ref is called. This function is in c-typeck.c file. The arguments passed to this function is the id. number of the structure and symbol information of the member. We can get IDENTIFIER_POINTER macro to get the name of the structure.

Our Collecting Mechanism

To implement collection of symbol information, we made a module to save a pair of the id. number and symbol name and modified files, c-decl.c and c-typeck.c to collect id. number and symbol name. Since C language does not have unified hash table implementation, we made hash table to store mapping of an id. number and a symbol name. To get symbol name of the structure at the point where the member of it is used, we save the name of the structure at xref_tag function and used it at build_component_ref function. To pass the names, we used the hash table implemented by us (Figure 3.10). To enable the implementation, we should rewrite makefile of cc1 and cc1obj command to make our hash table library and our symbol

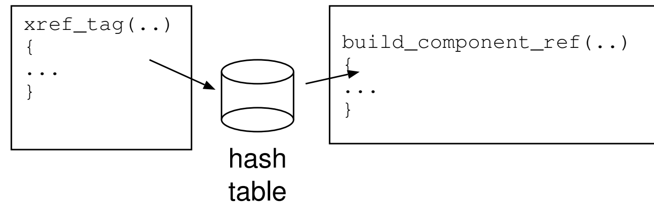

Figure 3.10: The Way to Map ID and the Name

storing library compiled with GNU C compiler program. Following explains the alternation of each function.

xref_tag Since the symbol information which contains the name of the structure is passed by parser, xref_tag function can get symbol name of the structure. The id. number of symbol information is also get by lookup_tag function. The we stores them into the hash table; the key of hash table is the id. number and the value of hash table is the name of structure.

build_component_ref Since id. number of the structure and the symbol information of the member of it is given to build_component_ref function by parser, all we have to do in this function is looking up the name of the structure and a number and the file name where the member is accessed. To get the name of the structure, we looks up the hash table by using the id. number as a key. If the name found, we store it to the file with the name of the member and the line number and the file name. The name of the member is get by using IDENTIFIER_POINTER macro. In our prototype implementation, the file to store symbol information is opened and closed every time we write symbol information to it.

To get the line number and the file name, we look up lineno variable and input_filename variable. Lineno variable contains the line number of the point parser parsing and input_filename contains the name of the file parser is parsing.[1] GNU C compiler stacks the file name and the line number if #include statement is read. The name of the stack is file_stack and we also store the stack information with the other data such as the names of the structure and the member to be able to find joinpoint which is inside the include file near future.

[1]These information is saved to input_location structure in GNU C Compiler version 3.4.x

3.2.3 Mechanism to Find Address

Since KLAS userland half can only get the line number and the file name of the point where the member of the structure is accessed, it have to resolve the address inside an OS kernel from them. To resolve address from the line number and the file name, we use DWARF2 information, which is used by GNU debugger to map a line number and an address of an instruction.

DWARF2 Information of ELF Binary

DWARF2(DWARF Version 2)[26] is one of the formats of debugger information of ELF format[27, 1]. GNU C compiler put the information for debug use in this format by default and we made parser of this format. ELF format is famous binary execution format of Unix and Unix-like operating systems. A ELF format file is composed with some sections and debugger information is stored to several number of these sections. The section named .debug_info stores the information of whole debug information. This section has information of each compilation unit and these information is stored in debugging information entries. This section uses .debug_abbrev section to show the length of debugging information entries. Compilation unit is used to be object file and is is linked to the ELF format file. Compilation unit is made from only source file("*.c" file) and these contains header files ("*.h" files) in it.

Following tells the way to parse information inside each section. Debug_info section contains the blanket information of debugging information and this section works with .debug_abbrev section which contains information each unit in .debug_info section, of the attribute and length. Debug_line section is assigned by .debug_info section to show mapping line and address of each compilation unit.

.debug_info section The debugging information of DWARF2 is in .debug_info section of ELF format file. This section is consist of a set of debugging information entries describing information about compilation unit. Each debugging information entry is described by identifying tag and contains a series of attributes. These are related to .debug_abbrev section to distinguish each attribute type and kind of information stored to debugging information entry of each compilation unit (Figure 3.11).

Each attribute value is characterized by an attribute name. The available values for an attribute is belong to one or more classes of attribute value forms. Each form class may be represented in one or more ways. Some

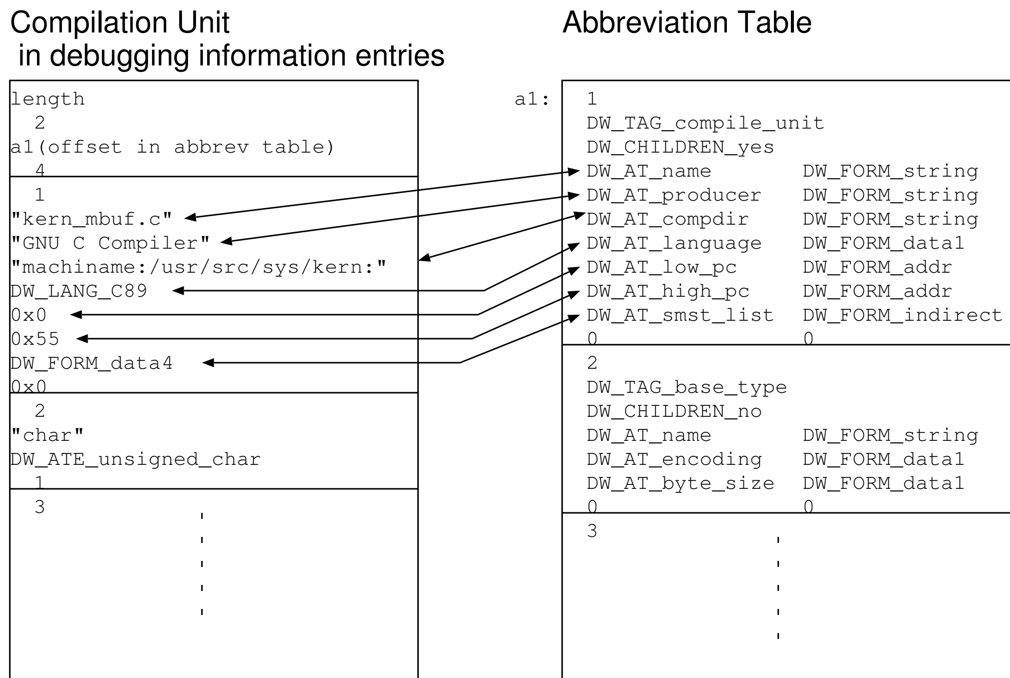

Figure 3.11: Organization of Debugging Information

attribute values, for example, consist of a constant data. "Constant data" is one of the class of attribute value. There are, however, several way to represent constant data: one byte, two bytes, four bytes and eight bytes, and variable data length. The particular representation for any given instance of an attribute is encoded along with the attribute name as part of the attribute name as part of the information that guides the interpretation of a debugging information entry. Classes attribute belong to is following.

address Refers to some location in the address space of described program.

block An arbitrary number of bytes of data.

constant One byte, two bytes, four bytes, eight bytes of data, or data encoded in the variable length format, LEB128.

flag A small constant that indicates the presence or absence of an attribute.

reference Refers to some member of the set of debugging information entries. There are two types of reference. One is an offset relative to the beginning of the compilation unit in which the reference occurs and must refer to an entry within that same compilation unit. The other is the address of any debugging information entry within the same executable or shared object; it may refer to an entry in a different compilation unit from the unit containing the reference.

string A null-terminated sequence of characters. Data in this form are generally printable strings. Strings may be represented directly in the debugging information entry or as an offset in a separate string table.

LEB128 is the variable length format of number used in DWARF2. Decoding algorithm is written in Figure 3.12 (unsigned) and Figure 3.13 (signed).

There are no limitations on the ordering of attributes within a debugging information entry. To prevent ambiguity, however, only one attribute with a given name may appear in any debugging information entries.

Information of each compilation unit is useful for getting a map of address and line. An object file may be derived from one or more compilation units. Typically a compilation unit represent the text and the data of executable binary from a single relocatable object and it may be derived from several source files including pre-processed include files. Each such compilation unit will be described by a debugging information entry with the tag DW_TAG_compile_unit.

The attributes compilation unit entry may have is following:

```
result = 0;
shift = 0;
while (true) {
    byte = next byte in input;
        result |= (low order 7 bits of byte << shift);
    if (high order bit of byte == 0)
        break;
    shift += 7;
}
```

Figure 3.12: Decode unsigned LEB128 number

```
result = 0;
shift = 0;
size = no. of bits in signed integer (32 in Intel x80 architecture);
while (true) {
    byte = next byte in input;
    result |= (low order 7 bits of byte << shift);
    shift += 7;
    if (high order bit of byte == 0)
        break;
}
if ((shift < size) && (sign bit of byte is et))
    result != - (1 << shift);
```

Figure 3.13: Decode signed LEB128 number

DW_AT_low_pc The relocated address of the first machine instruction generated for compilation unit.

DW_AT_high_pc The relocated address of the first location past the last machine instruction generated for that compilation unit.

DW_AT_name A null-terminated string containing the full or relative path name of the primary source file from which the compilation unit was derived.

DW_AT_language A code indicating the source language of the compilation unit. DW_LANG_C89 means ISO/ANSI C and DW_LANG_C means Non-ANSI C, such as K&R.

DW_AT_smst_list A reference to line number information of this compilation unit. The value of this attribute is the offset in the .debug_line section of the first byte of the line number information for this compilation unit.

DW_AT_macro_info A reference to the macro information for this compilation unit. The value of macro information attribute is the offset in the .debug_macinfo section of the first byte of the macro information for this compilation unit.

DW_AT_comp_dir A null-terminated string containing the current working directory of the compilation command that produced this compilation unit in whatever form make sense for the host system. Format of this value in Unix systems is "hostname:pathname". If hostname is not available, ":pathname".

DW_AT_producer A null-terminated string containing information about the compiler that produced the compilation unit. The actual contents of the string will be specific to each producer. The value of this attribute should begin with the name of the compiler vendor or some other identifying character sequence to avoid confusion with other producer values.

Each compilation unit is in .debug_info section and consists of a compilation unit header followed by a series of debugging information entries. Each debugging information entry begins with a code that represent an entry in a separate abbreviation table, table inside debug_abbrev section. This code is followed by a series of attribute values. Each compilation unit is associated with the proper entry of abbreviation table (Figure 3.11).

Each compilation unit has a header to represent itself. This is consist of following information:

1. A 4-byte unsigned integer represent the length of the information of compilation unit. This field does not include the length field itself.

2. A 2-byte unsigned integer represent the version of DWARF information of the compilation unit. For DWARF2, the value of this field is 2.

3. A 4-byte unsigned integer represent an offset inside the abbreviation table, .debug_abbrev section.

4. A 1-byte unsigned integer represent the size in byte of an address on the target architecture. In 80x86 architecture, our target architecture, the value of this field is 4.

Debugging information entries begins next to the header of the compilation unit. Each debugging information entry begins with an unsigned LEB128 number which represent an entry within the abbreviation tables in debug_abbrev section associated with this debugging information entry.

The abbreviation tables is shared by multiple compilation units. The abbreviation table for a single compilation unit consists of a series of abbreviation declarations. Each declaration consist of the tag and attributes for a form of debugging information entry and begins with an unsigned LEB128 number representing the abbreviation code itself. The same code appears at the beginning of a debugging information entry. In abbreviation code, the code '0' is reserved and it represent null debugging information entries. The abbreviation code is followed by another unsigned LEB128 encoded code and it represent the encode of the entry's tag.

Following tag encoding is a 1-byte value and it represent whether debugging information entries has a child or not. If the value is DW_CHILDREN_yes, next debugging information entry is a child of this entry. If it is DW_CHILDREN_no, the entry next to this is brother of this entry. Each chain of brother entry will terminated with a null entry.

Children encoding is followed by a series of attribute specifications. Each specification consists of two parts. The first part is an unsigned LEB128 format value which represent the name of the attribute. The second part is an unsigned LEB128 format value which represent the format of the data this entry presents. The series of attribute specifications finishes with entry containing 0 for the name and 0 for the format of the data. The format of the data is represented in table 3.1. Class of each format is defined before

Format	Class
DW_FORM_addr	address
DW_FORM_block	block
DW_FORM_block1	block
DW_FORM_block2	block
DW_FORM_block4	block
DW_FORM_data1	constant
DW_FORM_data2	constant
DW_FORM_data4	constant
DW_FORM_data8	constant
DW_FORM_string	string
DW_FORM_flag	flag
DW_FORM_sdata	constant
DW_FORM_strp	string
DW_FORM_udata	constant
DW_FORM_ref_addr	refference
DW_FORM_ref1	refference
DW_FORM_ref2	refference
DW_FORM_ref4	refference
DW_FORM_ref8	refference
DW_FORM_ref_udata	refference
DW_FORM_ref_indirect	indirect

Table 3.1: Attribute format encodings

Term	description
state machine	The hypothetical machine used by a consumer of the line number information to expand the byte-coded instruction stream int a matrix of line number information.
statement program	A series of byte-coded line number information instructions which represents one compilation unit.
basic block	A sequence of instructions that is entered only at the first instruction and exited only at last instruction.
sequence	A series of contiguous target machine instructions.
sbyte	1-byte signed integer.
ubyte	1-byte unsigned integer.
uhalf	2-bytes unsigned integer.
sword	4-bytes signed integer.
uword	4-bytes unsigned integer.
LEB128	Variable length signed and unsigned integer described before.

Table 3.2: Terms of Line Number Information Format

and a number followed by the name of the format is the size of the format: 2, for example, means 2 bytes. The class indirect did not represented before. If the class is indirect, the attribute value itself in the .debug_info section begins with an unsigned LEB128 number that represent its format.

By using these information, we can distinguish the file name and the directory name of each compilation unit and the offset of the line information of it. These information are enough for mapping an address and a line of files which composes the OS kernel.

.debug_line section This section represent the line number information, mapping of the line and the address of each compilation unit. The line number information is written in a byte-coded language, a kind of machine language. To decode this section, we should execute instructions of machine language of the region of this section where we would like to find out mapping. Offset is set for each compilation unit and it is described in DW_AT_smst_list attribute of .debug_info section.

Table 3.2 describes the terms used in the line number information. The statement machine represented in table 3.2 has the following registers:

address The value of program-counter which corresponds to a machine instruction.

register	value
address	0
file	1
line	1
column	0
is_stmt	determined by default_is_stmt value in the statement program prologue
basic_block	false
end_sequence	false

Table 3.3: Initial State of State Machine

file An unsigned integer indicating the id. number of the source file which corresponds to a machine instruction.

line An unsigned integer indicating a source line number. The value 0 means no line is corresponding to a machine instruction.

column An unsigned integer indicating a column number within a source line. The value 0 means that a statement begins at the "left edge" of the line.

is_stmt A boolean indicating that the current instruction is the beginning of a statement.

basic_block A boolean indicating that the current instruction is the beginning of a basic block represented in table 3.2.

end_sequence A boolean indicating that the current address is that of the first byte after the end of a sequence of target machine instruction. If this register is true, machine will stop executing.

At the beginning of execution of each machine instruction, the state of the register is table 3.3 value.

There are tree categories instructions of the state machine belongs to. Each categories are following:

special opcodes These consist of only a ubyte opcode field. The value of a ubyte opcode is always greater than opcode_base given in statement program prologue.

standard opcodes These consist of a ubyte opcode followed by zero or more LEB128 arguments.

extended opcodes These consist of multi-byte opcode and multi-byte argument. The first byte of this opcode is zero and next bytes are an unsigned LEB128 integer representing the opcode of the instruction.

The statement machine program has prologue before starting program body. The prologue show the length of instructions and version number and so on. The component of the prologue is as follows:

1. An uword integer represent total length of the state machine instruction (total_length).
2. An uhalf integer represent version identifier for the state machine instruction (version).
3. An uword integer represent length of this prologue (prologue_length).
4. An ubyte integer represent the smallest target machine instruction (minimum_instruction_length).
5. An ubyte integer represent the initial value of is_stmt register (default_is_stmt).
6. An sbyte integer used in special opcode (line_base).
7. An ubyte integer used in special opcode (line_range).
8. An ubyte integer represent the first value of special opcode (opcode_base). The opcode more than or equal to this value is special opcode.
9. An array of ubyte integer represent operands of each standard opcode (standard_opcode_length). The first element is associated with the first standard opcode. The length of this array is opcode_base - 1.
10. An sequence of string represent the pathname of include directories (include_directories). This sequence ends with two null bytes.
11. An sequence of string and an unsigned LEB128 integer represent the file name of each source codes composing this compilation unit (file_names). Each entry has a null-terminated string representing a file name and an unsigned LEB128 integer representing a id. number of include directory given in include_directory.

```
adj_opcode = opcode - opcode_base;
address_increment = (adj_opcode / line_range)
            * minimum_instruction_length;
line_increment = line_base + (adj_opcode % line_range);
```

Figure 3.14: Parsing a Special Opcode

Special opcodes of the statement machine program is parsed as figure 3.14 to get line_increment and address_increment values. Special opcode also have following effect on the state machine:

1. Add a line_increment value to line register.
2. Add a address_increment value to address register.
3. Record the value of line register and address register.
4. Set basic_block register false.

The value of line register can be decrement if the machine instruction compiler produced is for pipelined architecture. The value of address register, however, must not decrement.

Standard opcodes of the statement machine has following opcode:

DW_LNS_copy Record line register and address register, and set basic_block register false.

DW_LNS_advance_pc Add a multiplied value of a following unsigned LEB128 integer and minimum_instruction_length to address register.

DW_LNS_advance_line Add a following signed LEB128 integer to line register.

DW_LNS_set_file Set a following unsigned LEB128 integer to file register.

DW_LNS_set_column Set a following unsigned LEB128 integer to column register.

DW_LNS_negate_stmt Set logical negation of current value to is_stmt register.

DW_LNS_set_basic_block set basic_block register true

DW_LNS_const_add_pc Add a $\frac{minimum_instruction_length \times (255 - opcode_base)}{line_base}$ value to address register.

DW_LNS_fixed_advance_pc Add following uhalf integer to address register.

There are only 9 opcodes defined as standard opcodes and opcode_base value should be greater than 10.

Extended Opcodes begins with zero and followed by variable length opcodes. These value is following:

DW_LNE_end_sequence Just set end_sequence register to true and record address register and line register.

DW_LNE_set_address set address register to the following value. The value will change according to the architecture and can be hold enough size of each address. In 80x80 architecture, this size should be 4 bytes.

By executing statement machine instruction before on virtual statement machine, we can get a map of address and line number. The difficulty is only the way to get each section of ELF format file.

Our Implementation to Get an Address of a Line Number

Our resolver of address from number runs as daemon and used with interprocess communication. The aspects the user wrote is parsed and passed with this interface. The weaver passes the file name and the line number of pointcut and the daemon returns the address of the point the weaver suggested. The return value is formated with hex number.

Parsing Debugging Information We used BFD library to get debug_info section, debug_abbrev section and debug_line section. This is a library used by gcc and gdb and the license of this library is GPL. We used this library to manage ELF format file, especially to get sections described before.

BFD library provide us bfd_get_section_by_name function to find the offset of proper section inside ELF format file. We used this function and find each section. Then we used bfd_seek function to move file pointer and then used bfd_bread function to get the section contents. We did this operation for each sections.

Before use BFD library, we should execute bfd_openr function to open ELF format file. Then we should call bfd_check_format function to activate the opened file. To make reading first, we also used bfd_set_cachable function.

After reading each section, we parsed debug_info section to get information about each compilation unit. Information of compilation unit contains the file name of main source file composing this compilation unit. At the time parsing the section, we push each compilation unit to the hash table by using the file name of main source code as a key. We would like not to degrade speed of finding the address in weaving, we do not parse debug_line section until user request us to do so.

Parsing Line Section When the weaver requested the address from the file name and the line number, we start parsing from the offset of debug_line section, which the weaver give the name is associated to. Since the offset is parsed before and stored in hash table, we can look up the table and easily find the offset from the file name. We start executing the statement machine from the offset of debug_line section. We got mapping of addresses and lines. Then we return the address which is associated with the line the weaver required.

3.2.4 Parsing Aspects and Weaving Mechanism

Our aspect parser using XML parser of JDK since our aspect language is written in XML format. The parser makes object of each aspect and each aspect has the list of object representing pointcut and the object representing advice body. The addresses of the pointcut inside the object is resolved by the operation described before. KLAS insert hooks to these addresses and and advice body is associated with them (Figure 3.15).

The object for pointcut has a polymorphism structure and is in the list of pointcut. To use the pointcut object, we should distinguish which kind of pointcut it is. If the object is function type pointcut, we call nm command and look up the address of joinpoint. If the object is structure type pointcut, we first find out the list of the file name and the line number where the member of the requested pointcut exist by using extra symbol table, and then we find out the addresses of these member access by using mapping of the address and the line.

Both the module which resolve the line number and the file name from the member name and the structure name and the module which resolve the address from the line number and the file name are ran as daemon. To find out the addresses of pointcut, we should communicate first module to get the line number and the file name, and then second module to get the addresses (Figure 3.16). After finding the address, KLAS userland half first call kldload command to load advice body into OS kernel and then call a

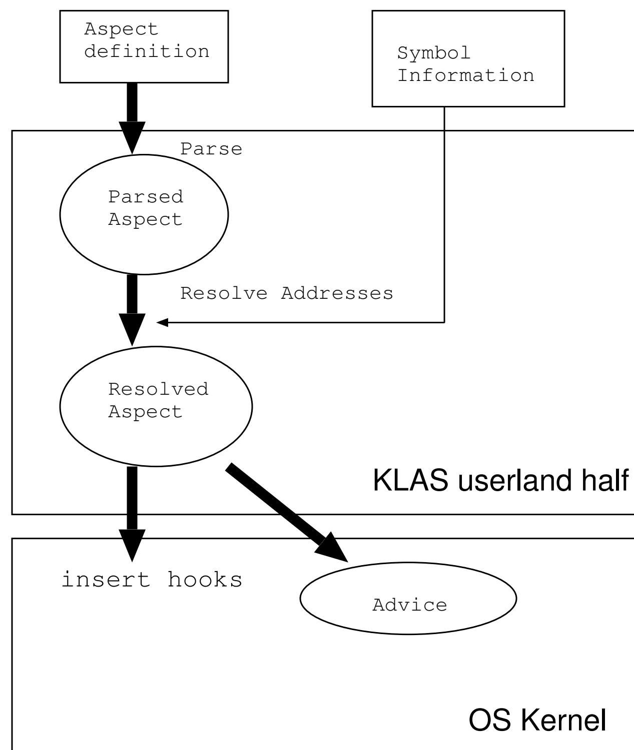

Figure 3.15: Action of KLAS Userland Half

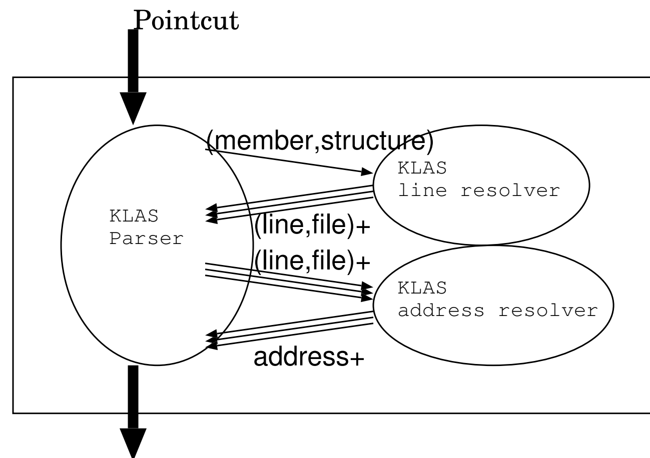

Figure 3.16: Action to Resolve Addresses

system call to let KLAS kernel half to insert hooks into the OS kernel and associate them with the advice body (Figure 3.3).

Chapter 4

Experiment

4.1 Accuracy of Positions of Hooks

A target of our system KLAS is evaluation of performance of code snippet where the users specified. The users will measure the time at the point where a function starts execution and at the point where a member access of structure occurs. The users must need accuracy of a point to insert hook where a member access occurs and a point where a function is executed and error of the position to insert hooks should be minimized.

Nonetheless, since our system, KLAS, uses a line numbers and a file name to find out a address to insert hook, KLAS can not accurately insert a hook to member access of structure which occurs middle of a line. KLAS produce symbol information with not program counter grained but line grained. That is because of our implementation and we should reform this point to be program counter grained accuracy. If the member is a function pointer, however, the line should be begin with the structure. The point to insert hook should be accurate. Even if the member is not a function pointer, the problem is only the point where the member access occurs in the same line of function call and it might not be so many.

Our system can not insert hooks to the point where the instructions comes from *include file* either. Line information of each *include file* can be get when the main source file of each compilation unit is needed. That is because KLAS use debug information in lazy way not to delay start up of the KLAS system and not to use memory so much. KLAS do not parse line information of each compilation which has a information of include files until the users want KLAS to look up the line of the main source code of the compilation unit. This problem can be solved easily to let KLAS find out

relationship between main source codes and include file at compile time.

4.2 Simple Performance Measurement

We evaluate our design proposals on an 1800 MHz AMD Athlon(TM) XP 2200+ configured to use 1024MB of memory, running our modified FreeBSD 5.2.1 kernel.

4.2.1 Performance Measurement of Kernel Compile

Compile time of OS kernel should be slow by using our modified compiler. We compared elapsed time of compiling OS kernel between original C compiler, GNU C compiler we do not modified and our modified C compiler. We set "NO_MODULE=yes" option into make.conf not to make kernel modules and we used GENERIC configuration file for compiling OS kernel. Time measurement is done by using Unix time command.

The result of this experiment is showed at table 4.1. We measured tree times and calculated average of real time and system time and user time. Real time means the whole elapsed time of compiling OS kernel and system time means elapsed time only inside OS kernel and user time means elapsed time only inside userland. According to the result, Our compiler is about 1.8 times slower than the ordinal C compiler in real time. It can be say that this is meaningful difference between our modified compiler and normal one. Most of time is used not inside userland but inside system. Difference of user time is much smaller than difference of system time. Since our modified compiler open and close the file for writing and flushing symbol information to the file wherever member access found, Cause of this result must be the implementation of our modified compiler. Overhead might be smaller by caching file handler throughout compiling and we should implement the overhead light. The total elapsed time of our modified compiler, however, should be much smaller than compiling OS kernel whenever the users want to change codes or change That is because the KLAS system needs only one time for compiling and no more compile needed for changing codes and points.

4.2.2 Performance Measurement of Execution

To examine the execution overhead of KLAS, we measured the execution time of the kernel codes by increasing a number of points to insert hooks. For experiment, we made a system call which call ten thousand of null body

kind of compiler	real ave.	user ave.	sys ave.
normal gcc	294.25	211.72	79.54
modified gcc	527.86	223.06	301.07

Table 4.1: Elapsed Time for Compiling OS Kernel

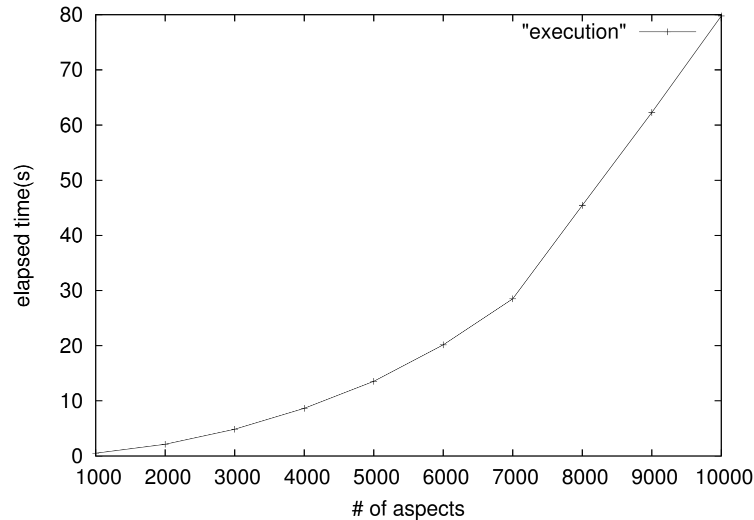

Figure 4.1: Elapsed Time by Increasing a Number of Aspects

functions. We measured the elapsed time of the execution of the system call by changing a number of the aspects which are weaved to these functions. We weaved only an aspect into a function to make effects of aspects clearly.

The results of experiment is described in figure 4.1. This says the performance degrade according to a number of aspects. It looks increasing linearly from 7000 to 10000 and looks exponentially from 1000 to 7000. We also printed log-scale graph of figure 4.1 to distinguish elapsed time increase exponentially or not (Figure 4.2). A slope of elapsed time degrade by increasing a number of aspects. The equation of elapsed time might be linearly and not to increase exponentially.

A number of aspects, however, should not be so large when profiling OS kernel. We also measured the elapsed time with smaller number of aspects (Figure 4.3). According to the result, it looks that increase of elapsed time is exponentially. We printed the same graph with log scale to distinguish the

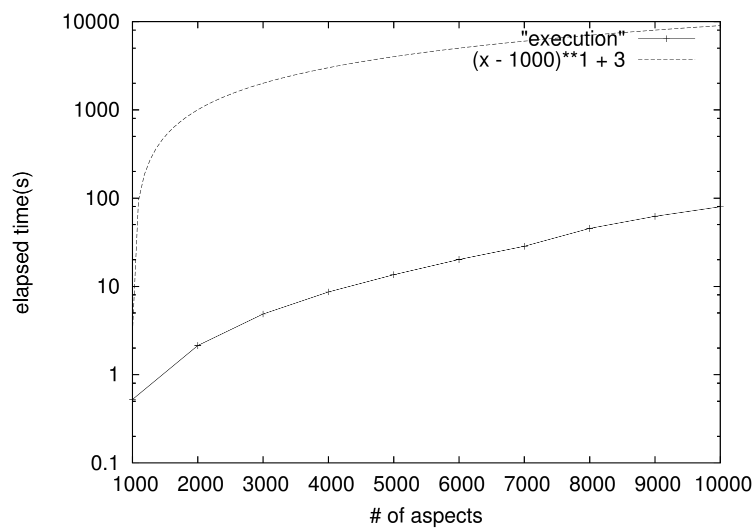

Figure 4.2: Elapsed Time by Increasing a Number of Aspects (Log-scale)

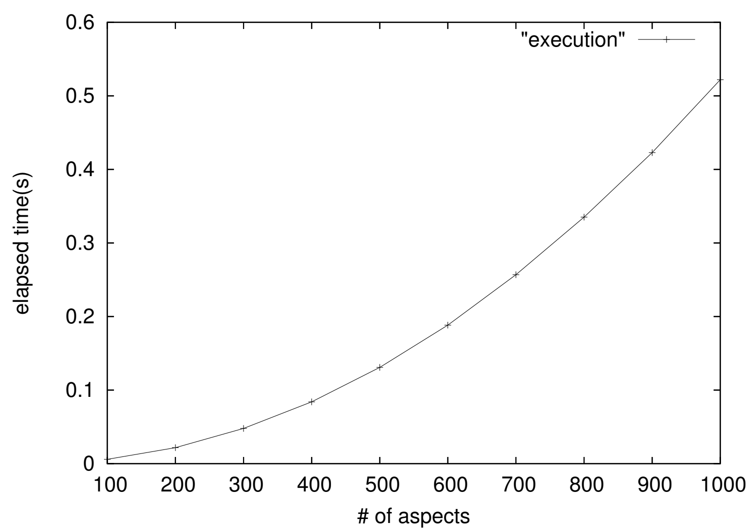

Figure 4.3: Elapsed Time with a Smaller Number of Aspects

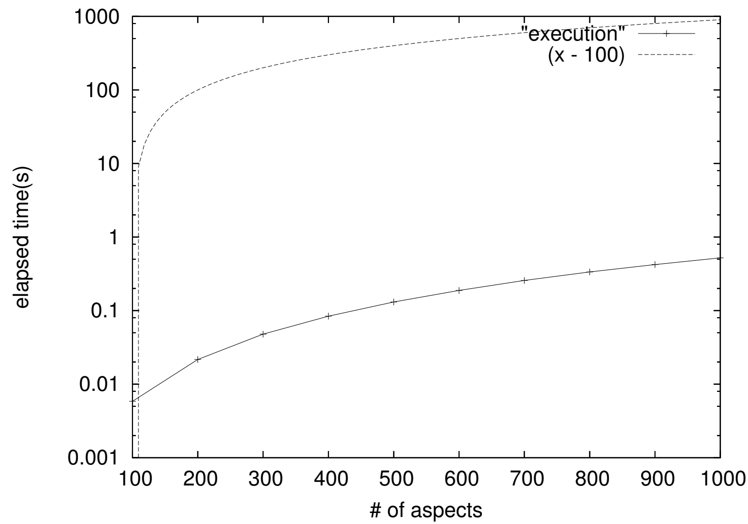

Figure 4.4: Elapsed Time with a Smaller Number of Aspects (Log-scale)

increasing is exponentially or not (Figure 4.4). A slope of elapsed time also degrade by increasing a number of aspects and it might be not exponentially increasing but linearly increasing.

4.3 Performance Measurement of Network Codes

We also evaluate our design proposals on an 1800 MHz AMD Athlon(TM) XP 2200+ configured to use 1024MB of memory and 100Base-TX Intel 82550 Pro/100 Ethernet Adaptor, running our modified FreeBSD 5.2.1 kernel. The host communicating with machine with our modified FreeBSD kernel is the same machine specification and running FreeBSD 5.3 kernel. Measurement is done by sending packets from the machine with FreeBSD 5.3 kernel to the machine with our modified FreeBSD 5.2.1 kernel. We used `netstat` command to watch throughput of the communication.

We measured the throughput of TCP/IP network by inserting aspects into network codes of OS kernel. First, we inserted aspects which have null advice to whole TCP functions, functions whose name has "tcp" string in prefix. Then we inserted the aspects to whole IP functions, functions whose name has "ip" string. Finally, we inserted aspects into both TCP and IP functions and measured throughput.

Kind of Advice	TCP	IP	Both
Null Advice	98.5Mbps	98.3Mbps	48.6Mbps

Table 4.2: Throughput Change of Inserting Advice

The result of the experiment is described in table 4.2. According to the result, the overhead of KLAS is not so big not to weave aspects both IP and TCP codes. The reason of performance degradation might be a latency of TCP time out of sliding window. As aspects increase, a number of breakpoint-traps increase and inserting both aspects over the threshold of permissible range of latency of TCP.

Chapter 5

Related Work

5.1 Kernel Profilers

5.1.1 Ktr

Ktr[15] 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. The syntax of CTRx macros is like a printf function and the users can easily use these 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. Each logs are stored to ring buffer in kernel memory at runtime and the user can move them to disk by ktrdump command.

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 by accessing ktr_mask. This approach, however, implies non-negligible overhead.

5.1.2 LKST

LKST[10, 9] 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. LKST covers most events which are often the cause of bugs. Since events are stored to disk, the users can use the system for debugging after panic of OS kernel.

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 might be a good way to debugging OS kernel. This feature, however, makes it difficult to investigate the behavior of the kernel in a fine-grained way. That is because the users can not shorten the range of the section they want to find performance bottlenecks. As we mentioned before, shortening the range is needed to find out performance bottlenecks efficiently.

5.1.3 KernInst and GILK

KernInst[24] and GILK[19] can dynamically transform the binary code of the OS kernel. It has been successfully applied to kernel performance measurement and runtime optimization. The Linux version of KernInst use the same way with KLAS kernel half to insert codes into running OS kernel. GILK uses **jmp** instruction to insert codes into OS kernel.

The users, however, 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. Moreover the space available for code patches and associated data is very limited and weaving a large number of aspects is not suitable for them. In addition, GILK can not insert codes into the section which is not basic block. KLAS can insert codes into the section instead because KLAS use a breakpoint-trap for a hook.

5.2 Other Aspect-oriented Solutions

5.2.1 AspectC

AspectC[7, 3, 12, 6, 5, 4] 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. 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 join points.

5.2.2 PROSE

PROSE[20] 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 join points 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 join point.

This idea is the same as ours but KLAS is a dynamic AOP system for the OS kernel written in C. The cost of breakpoint traps in the OS kernel is relatively smaller than in the JVM.

5.2.3 Wool

Wool[21] is another dynamic AOP system for Java. The implementation of Wool is a hybrid of two implementation techniques. At first, Wool sets a breakpoint at the join point picked out by a pointcut. Then, if the thread of control frequently reaches that join point, Wool changes the implementation. It removes the breakpoint and reload the modified bytecode in which the advice body is embedded. This hybrid approach improves total execution performance.

Wool can perform this hybrid approach since the binary code of a Java program includes richer symbol information than in the C language. To obtain as rich symbol information in C as in Java, we have extended a C compiler for KLAS.

Chapter 6

Concluding Remarks

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 thesis, 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.

Experiment shows that our modified C compiler spent much time compared with normal C compiler. Since most of time is elapsed at system space, we should decrease a number of file I/O to decrease overhead of compiling. Experiment also represent that execution overhead of KLAS is much smaller when a number of aspects is small. However, experiment also showed that overhead becomes much larger when KLAS runs with a quite large number of aspects.

Bibliography

[1] A WIKIMEDIA Project: *Executable and Linkable Format*, Online Publishing, URI `http://en.wikipedia.org/wiki/Executable_and_Linkable_Format`.

[2] Buck, B. and Hollingsworth, J. K.: An API for Runtime Code Patching, *The International Journal of High Performance Computing Applications*, Vol. 14, No. 4, pp. 317–329 (2000).

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

[4] Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., Ong, J. S. and Gudmundson, S.: *The 8th Workshop on Hot Topics in Operating Systems (HotOS)* (2001).

[5] Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., Ong, J. S. and Gudmundson, S.: Exploring an Aspect-Oriented Approach to OS Code, *4th ECOOP Workshop on Object-Orientation and Operating Systems* (2001).

[6] Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N. and Ong, J. S.: Structuring Operating System Aspects, *Communications of the ACM (CACM)* (2001).

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

[8] Eclipse Organization: *aspectj project*, Online publishing, URI `http://www.eclipse.org/aspectj/`.

[9] Hitachi,Ldt.: LKST（Linux Kernel State Tracer) - A tool that records traces of kernel state transition as events., Online publishing, URI `http://oss.hitachi.co.jp/sdl/english/lkst.html`.

[10] Hitachi,Ltd. Fujitsu,Ltd: *Linux Kernel State Tracer*, Online publishing, URI `http://lkst.sourceforge.net/` (2001).

[11] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, Technical Report SPL97-008 P9710042, Xerox PARC (1997).

[12] Kiczales, G. and Coady, Y.: *AspectC*, Online publishing, URI `http://www.cs.ubc.ca/labs/spl/projects/aspectc.html`.

[13] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *Lecture Notes in Computer Science*, Vol. 2072, pp. 327–355 (2001).

[14] Leavens, G. T. and (eds.), C. C.: FOAL 9,003 Proceedings - Foundations of Aspect-Oriented Langauges Workshop at AOSD 2003.

[15] Lehey, G.: Improving the FreeBSD SMP Implementation, *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, USENIX Association, pp. 155–164 (2001).

[16] McKusick, M. K., Bostic, K., Karels, M. J. and Quarterman, J. S.: *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Longman,Inc. (1996).

[17] McKusick, M. K. and NEVILLE-NEIL, G. V.: *The Design and Implementation of the FreeBSD Operating System*, Addison-Wesley Longman,Inc. (2004).

[18] OS Research Group at the University of Magdeburg: *The PURE Manipulator*, Online publishing, URI `http://ivs.cs.uni-magdeburg.de/~puma/`.

[19] Pearce, D. J., Kelly, P. H. J., Field, T. and Harder, U.: GILK: A Dynamic Instrumentation Tool for the Linux Kernel, *Computer Performance Evaluation / TOOLS*, pp. 220–226 (2002).

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

[21] Sato, Y., Chiba, S. and Tatsubori, M.: A selective, just-in-time aspect weaver, *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, Springer-Verlag New York, Inc., pp. 189–208 (2003).

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

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

[24] Tamches, A. and Miller, B. P.: Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels, *Operating Systems Design and Implementation*, pp. 117–130 (1999).

[25] the AspectC++ project: *The Home of AspectC++*, Online publishing, URI `http://www.aspectc.org/`.

[26] TIS Committee: *Tool Interface Standard(TIS) DWARF Debugging Information Format Specification Version 2.0*, Online Publishing, URI `http://www.x86.org/ftp/manuals/tools/dwarf.pdf` (1995).

[27] TIS Committee: *Tool Interface Standard(TIS) Executable and Linking Format(ELF) Specification version 1.2*, Online publishing, URI `http://www.x86.org/ftp/manuals/tools/elf.pdf` (1995).