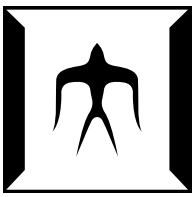

A Dissertation Submitted to Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of Doctor of Science in Mathematical and Computing Sciences

---

A Dynamic Aspect-oriented System for Data-driven Profiling of OS Kernels

データ主導のプロファイリングのための カーネル用動的アスペクト指向システム

Yoshisato YANAGISAWA

Dissertation Chair:
Shigeru CHIBA

February 2008,

Abstract

We propose a data-driven dynamic aspect-oriented system for profiling OS kernels. A dynamic aspect-oriented system can change a running OS kernel without recompiling and rebooting it. This improves efficiency of development since the developers can avoid waiting until some problems occur again. The ability to change the execution points and code is crucial. Developers 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. As the investigation continues, they will also change interesting data for getting a log.

For profiling OS kernels, we added the access pointcut and the xflow pointcut to a dynamic aspect orientation for the C language. Since the C language does not have a language mechanism to make a module like a package or a class in an object-oriented language, the access pointcut is important. It selects a member access of a structure as a join point. We used a technique named source-based binary-level dynamic weaving for implementing this feature. The technique collects richer symbol information at compile time to use it at run time for getting a memory address of a member access. The xflow pointcut ensures a profiling based on a data flow. Since an OS kernel is a multi-thread application, profiling it by tracing a control flow is difficult. Developers can specify start, propagate, and quit points of a data flow in details. They can avoid logging unnecessary data by using the xflow pointcut.

We implemented those pointcuts on Linux and Xen, and thereby showed that our ideas can be implemented with acceptable performance. For the access pointcut, we experimented with the UnixBench benchmark, and confirmed that the overheads are acceptable in most case. For the xflow pointcut,

we did micro benchmarks on functions that are used for implementing the pointcut, and confirmed that overheads of those functions are acceptable. We also did some case studies, and it showed that those pointcuts are useful for real profiling.

Acknowledgments

I would like to express my deep gratitude to my supervisor, Shigeru Chiba. He has supported me for 6 years from a bachelor student to a PhD student in Tokyo Institute of Technology. He thought me various things, from the way of studying to the way of presentation. He also provided me a great environment to study, from laboratory equipment to relationship in the laboratory. Kindness he gave me is too numerous to comprehensively list here.

I also deeply thank Kenichi Kourai, who has been a research associate of Tokyo Institute of Technology since I was a first year master student. He gave me various important ideas and advice: the way of improvement and implementation of software, and the way of writing a paper and presentation, and so on, from a basic level to a high level. He also greatly encouraged me in my studies; especially at the time our paper was rejected by one conference after another. He gave me his precious time freely to me. I profoundly thank him again.

Moreover, I thank my colleagues of the Chiba Shigeru Group (aka. CSG) in Tokyo Institute of Technology: Muga Nishizawa, Romain Lenglet, Yoshiki Sato, Daisuke Yokota, Ryo Kurita, Kiyoshi Nakagawa, Masahiro Matsunuma, Rei Ishikawa, Hideaki Hibino, Yoshiyuki Usui, Yasuhiro Aoki, Natsuko Kumahara, Hideyuki Takeuchi, Yohsuke Kurita, Yuji Takizawa, Michihiro Horie, Ryunosuke Imayoshi, Aya Uchikawa, Hidekazu Tadokoro, Shumpei Akai, Takeshi Azumi, Satoshi Morita, and Masayuki Ioki. Especially, Yoshiki Sato gave me a kick in the back to go to a Ph.D program, Romain Lenglet gave me various important advice on my studies, Rei Ishikawa helped me to write some programs, Michihiro Horie gave me some advice on my thesis, and discussion with Muga Nishizawa was quite interesting. I also thank secretaries at Tokyo Institute of Technology: Natsuko Katagai,

Kazuko Sakata, and Yuuko Tatsuzawa. They have been working behind the scenes, and sometimes gave me some important advice on my life as a PhD student.

At some workshops, Kenji Kono, Shuichi Oikawa, Hidehiko Masuhara, and Yvonne Coady gave me some important advice, and encouraged me in my studies. I thank them. I also thank PhD students and graduates in other laboratories: Yoshihiro Kanamori, Hideyuki Jitsumoto, Hitoshi Sato, Takefumi Miyoshi, Akiyoshi Sugiki, Hiroshi Yamada, Miyuki Hanaoka, Masato Asahara, and Miho Yanagisawa. Their existence always encouraged me for studying. I have been developed through friendly competition against them. I am grateful to my thesis committees. They reviewed the submitted version of this thesis and the final version reflects their comments. The committees are organized by Shigeru Chiba, Masataka Sassa, Osamu Watanabe, Satoshi Matsuoka, and Ken Wakita.

Finally, from the bottom of my heart, I want to thank my father (Nobuhiro), my mother (Mayumi), my brother (Tadahiro), and my friends, especially Yoko Shinozaki, for support and care during my years of studies. My parents put me a road to a geek-dom, and of course, put me through college. My mother encouraged me in my studies by saying, "Study seriously or quit quickly." My brother also put me a geek road when I was a junior high school student through showing me his interest in a computer.

This thesis was partly supported by the CREST program of Japan Science and Technology Corp.

Yoshisato Yanagisawa
February 28th, 2008

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Throughout the history of operating systems (OSes), performance improvement of systems has been always one of the most important topics. The first edition of Unix was developed because the performance of the Multics (Multiplexed Information and Computing Service) mainframe timesharing system was not scalable as its developers expected in 1969 [61, 44]. The fast file system (FFS) was developed for improving performance and reliability of the original 512-byte Unix file system in early 1980s [49].

Even now, performance improvement is an important topic. The technology of OSes is still immature. The ULE scheduler was developed for FreeBSD in 2002 [62], and a new version is currently being developed for using a new lock feature. New optimization techniques of file systems, which is known as soft updates [50], dirperf, vmiodir and dirhash, were also developed for improving the FreeBSD's FFS performance from late 1990s to early 2000s [23]. Soft updates were shown comparable to journaling in 2000 [66]. System call implementations of NetBSD and FreeBSD were changed due to the results of some benchmarks in 2003 [76]. Even in Linux, some schedulers have been proposed [54, 78] so far, and the default scheduler of Linux was changed from the O(1) scheduler to the Complete Fair Scheduler (CFS) in 2007 [39].

For improving OS performance, the developers first have to investigate performance bottlenecks. Profiling is generating logs of measured elapsed

time between a number of execution points. After profiling, the developers read logs generated by profiling to investigate the performance bottlenecks. Profiling is a research area that has a long history [31, 70, 24]. A naive and most popular way of profiling is manually editing several statements so that a timestamp will be printed when the thread of control reached one of those statements. However, there are too many execution points for developers to enumerate correctly. Moreover, editing statements by hand is error-prone. Developers tend to edit incorrect statements, and sometimes remove necessary codes by accident. The naive way needs recompiling and rebooting the OS kernel. Rebooting the OS kernel clears memory, which may contain trails of strange behavior that would take long time to happen again.

To avoid the problem of the naive approach, the developers should use a good profiling tool (aka. profiler). Since execution points for profiling are automatically selected by the profiling tool, the developers can avoid editing wrong statements by mistake. A famous profiling tool for userland applications is gprof [31]. It counts the number of execution and measures execution time of each function. For measurement, a program to be profiled should be compiled with some special option such as `-pg`. For kernel profiling, the developers should want to avoid recompiling and rebooting the OS kernel since it will lose a large amount of time. Thus, most kernel profilers do not require the developers to recompile and reboot the OS kernel for starting and stopping profiling. The developers can activate and deactivate profiling at run time. Moreover, some profilers allow the developers to choose what kind of profiling is activated and deactivated.

1.1 Kernel profiling with flexibility and abstraction

Those profiling tools for kernels dramatically improve efficiency of performance investigation. However, they still have a problem. They can provide a log of timestamps and values of some variables at arbitrary execution points related to what the developers want to investigate. Some profilers enable the developers to write an arbitrary code for profiling. This feature is useful for reducing memory space since the developers can avoid getting a log of unnecessary information. For the purpose, we propose our two requirements for a kernel profiler: flexibility and abstraction:

Flexibility A profiler should generate timestamps at arbitrary execution points in an OS kernel. In addition, it should enable the developers

to describe an arbitrary profiling code.

Abstraction A profiler should provide good view for selecting an execution point to get timestamps. In addition, it should allow the developers to write a profiling code in a high-level programming language.

A degree of flexibility is high if a profiler enables the developers to execute an arbitrary code snippet at an arbitrary execution point. This feature is required since the code snippets for profiling and execution points to get logs will change by going of profiling. The ability to change the execution points is crucial. The developers 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. Furthermore, the code snippets for measuring the elapsed time must be given by the developers since they 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 developers can give such code depending on particular use case. In addition, they may want to print a log message with the value of some interesting variables.

A degree of abstraction is high if a profiler enables the developers to specify execution points and code snippets for profiling with source-code level view. Here, we mean source-code level view as view that enables the developers to easily select source level symbols to execute profiling codes. This feature is required since the developers can easily specify code snippets and execution points if higher abstraction is provided. It will improve effectiveness of profiling. Ease of specifying execution points for profiling is important. If finding an execution point for inserting a code snippet is difficult, the developers will take a long time before insertion of profiling code snippets. Moreover, code snippets inserted into the execution points for profiling should have good abstraction since it will also improve effectiveness of profiling. If abstraction for a code snippet is assembly-level abstraction, they will also take too long time for writing a profiling code snippet before profiling and will get tired.

However, existing profilers are not fulfilling all our requirements. There are two kinds of profilers but neither of them fulfills both of our requirements. One is an event-driven method. Profilers using the method will get a log when a specified event occurs. Some this-type profilers can use arbitrary code for getting a log. Although their degree of abstraction is high, their degree of flexibility is quite low. The developers can execute profiling code only at the execution points that are specified by creators of profilers before compile

time. They are too little to do profiling effectively. More worse, this kind of profilers have some overheads even if developers do not do some profiling because pieces of code, which we call hooks, are inserted for calling an event handler at compile time. SystemTAP [59] and DTrace [12] are examples of this kind of profilers, and are known as useful profilers. They provide their own language and enable the developers to write profiling code in higher abstraction. However, execution points used for profiling are quite limited, for example the beginning of functions.

The other is a dynamic code instrument method. Profilers using the method can insert an arbitrary code snippet into an arbitrary execution point. They can insert a code snippet into an execution point specified with a memory address. Although their degree of flexibility is quite high, their degree of abstraction is low. They only provide assembly-level abstraction if we want to insert a code into an arbitrary execution point. Kerninst [71] and GILK [57] are famous examples of this kind of profilers. They can insert a code snippet into an arbitrary memory address of a running Linux kernel. However, they provide assembly-level abstraction for inserting a code snippet, and they are hard to use for daily profiling.

1.2 Approach by this thesis

To provide a profiler that fulfills both requirements for flexibility and abstraction, this thesis proposes using the idea of dynamic aspect-oriented programming (DAOP) for profiling. For flexibility, DAOP enables the developers to select various key execution points, such as function calls, function executions, and assignments of variables. For abstraction, it provides source-code level view. The developers can write a code snippet and specify an execution point to execute profiling code in higher abstraction. By DAOP, the developers can also change execution points and code snippets for profiling without rebooting an OS kernel. They can start profiling when they need profiling. Since code insertion can be done without rebooting the OS kernel, memory will not be cleared after inserting a code snippet for profiling.

Aspect-oriented programming (AOP) is a way of modularization. An aspect is composed of pointcuts and advices. Pointcuts select the execution points to execute profiling code. With a pointcut, the developers can select an execution of a function, a call of a function, an assignment of a variable, and so on. Advices are code executed at an execution point specified by a pointcut. Advices are usually written with a target language of an AOP

system. Composition of an aspect to the original code is called *weaving*, and decomposition is called *unweaving*. DAOP is a mechanism for dynamically weaving or unweaving an aspect at run time. With the mechanism, the developers can weave and unweave an aspect without rebooting an OS kernel. DAOP also has an abstraction that AOP has.

However, existing DAOP systems are insufficient for profiling. They do not have a pointcut to select a series of data. Since an OS kernel uses a member access of a structure to provide higher-level abstraction, we need a pointcut to select it. The pointcut will enable the developers to do efficient profiling. Similar pointcuts such as set and get pointcut exist in a DAOP system for Java. However, since most of commodity OS kernels are written in the C language, the developers cannot use it. Moreover, since a modern OS kernel is multi-threaded software, a pointcut that can select a data flow is required for profiling an OS kernel. One data is manipulated by multiple threads and modules for fast interrupt handling and delicate processing of complex data. Existing DAOP systems for the C language do not have appropriate pointcuts for selecting this kind of data. To solve the problems, we have added the access pointcut [80, 81] and the xflow pointcut [79] to a DAOP system for the C language.

1.2.1 The access pointcut

The access pointcut selects a member access of a structure. We made KLASY (Kernel Level Aspect-oriented SYstem) for implementing the access pointcut. Since a member access of a structure is used among related modules in an OS kernel, selecting it with a pointcut will reduce difficulty of profiling. If the developers do the same thing with execution pointcuts, which will select a function call, they need much knowledge for the OS kernel. Moreover, enumerating related functions need patience and it will take a long time even if they can use wild cards. According to our case study in Chapter 6, some functions cannot be selected with execution pointcut. That is because it is defined as static inline function, and eliminated at compile time. If the developers want to select those functions, they should disable optimization. However, an OS kernel such as FreeBSD and Linux are usually compiled with some optimization options, using an OS kernel without optimization for profiling is unrealistic. Even if the developers do profiling with such kernel, the results should be meaningless.

Our main contribution on the access pointcut is its implementation. We used the technique named source-based binary-level dynamic weaving. The

technique is that richer symbol information is collected at compile time, and then it is used at run time for getting a memory address of a member access. It is also used to get a register number or a memory address of local variables and function arguments. With this technique, our DAOP system can select various pointcuts that can be selected by a static AOP system. For collecting richer symbol information, we modified the GNU C compiler (aka. gcc). The modified compiler collects a line number and a file name of each member access. The modified compiler also collects a memory address of each line with enabling the debug option (-g). The information of a memory address for each line is called the line information.

Since we need the multiple line information for a code generated by the compiler, we modified the compiler and the GNU assembler (gas). A register transfer language (RTL) generator and the sub systems following the RTL generator, such as an RTL optimizer, of the compiler only remain the first line information by default. They omit exceeded line information if multiple line information is associated to a code. The assembler also remembers only the first line information, and omits exceeded information by default.

Remaining the line information is important because it helps searching a memory address of a line from the optimized OS kernel. Optimization sometimes shrinks multiple lines to one, and line information for those lines is associated to a code generated by optimization. To get a memory address of those lines, line information should not be reduced to one. Moreover, multiple line information is needed for ease of searching a memory address of a code. Since line information is made per a .c file, we need to know which .c file contains it when we get a memory address from a line number. However, only line information for a line in a .h file is remained, and line information for a line in a .c file is omitted at by default. It is difficult to find the memory address without remaining multiple line information per a code.

For getting local contexts, KLASY generates a trampoline function. With this feature, the developers can use a value of a local variable or a function argument in an advice. When the developers weave an aspect, its advice is compiled, and the compiled advice is loaded into the kernel memory. After that, a *hook* code is inserted to an execution point specified by a pointcut for calling the trampoline function. The trampoline function passes a reference to a local variable or a function argument to the compiled advice. Note that the *hook* code is executed instead of the original code when the thread of control reaches a specified execution point. Also note that the *hook* code pushes all registers to a stack before calling the trampoline function. For generating the trampoline function, KLASY investigates debug information

generated at compile time to know a register number or a memory address where the variable or the argument exists. After investigation, KLASY generates a special function to pass a reference to the stored register value or the memory address. The trampoline function is also compiled, and is loaded into the kernel memory.

1.2.2 The xflow pointcut

The xflow pointcut is used to find a structure instance that has data of a specified data flow. We made XenLASY by extending KLASY for implementing the xflow pointcut. Since a modern OS kernel is multi-thread software, a mechanism to finding data related to a specified data flow is useful. The developers can trace data from the beginning to the end. It is difficult to do the same thing by tracing a control flow (aka. cflow) or simply tracing a data flow. If the developers use a control flow, they cannot keep tracing after a change of threads managing data to investigate. The change of threads often caused in the OS kernel since data is always managed by two kinds of threads, the top half and the bottom half, for keeping low latency and achieving complex tasks. If the developers simply use a data flow, they cannot trace copied data of data to investigate. For example, data is copied at the TCP module in the OS kernel for retransmission processing of TCP.

The significant feature of the xflow pointcut is that the developers can specify the definition of the xflow pointcut. They can specify execution points to start tracing, change tracing data, and stop tracing. With this feature, the developers can avoid storing unnecessary data, and they can save memory space for storing logs. The developers can tell XenLASY how data is passed among threads, and XenLASY traces the data flow according to the definition given by the developers. Since the xflow pointcut is developed to work on the Xen virtual machine monitor, the developers can keep on tracing even if data is passed through one virtual machine (VM) to another. For this mechanism, the developers can trace a data flow even after a VM that manipulating data changes. Since each data flow has its own ID number, the developers can distinguish one data flow from others. This helps the developers to how each data is manipulated in detail.

XenLASY manages data flows by using a table. The table contains a pair of a pointer to a structure instance and its own ID number. The pair is stored in the table when tracing starts, and it is removed from the table when tracing ends. The developers specify which structure instance to use, and where to start or end tracing through the definition of the xflow pointcut.

Since the ID number is got from the pointer to the structure instance, the ID number can be got even after threads are changed. For keeping on tracing after change of the structure instance, the mapping between the ID number and the pointer is changed. The mapping between the ID number and the pointer is removed, and the mapping between the ID number and the new pointer is created according to the definition specified by the developers.

For keeping on tracing after VMs change, the ID number is passed to a destination VM with data. For passing the ID number, XenLASY stores it in the unused space of the header associated with the data. This is also specified by the developers by the definition of the xflow pointcut.

1.3 Position of this thesis

The position of this thesis in the research history of programming language is that it provides flexible profiling without ignoring abstraction. As illustrated in Figure 1.1, profilers and existing DAOP systems are difficult to satisfy the both requirements: flexibility and abstraction. That is because compiled binaries of C programs usually do not have symbol information enough for understanding source level view. Therefore, some systems focused on flexibility. They allow the developers to insert an arbitrary code into an arbitrary position. However, their abstraction is only the assembly level since those systems can only use poor symbol information, such as global variables and function names, because detailed information is eliminated at compile time. The developers should calculate the position by hand. They sacrifice abstraction for flexibility. On the other hand, other systems, such as profilers, provide good abstraction. The developers can select the position to print a log message with higher abstraction, source-code level view. However, a number of execution points that the developers can use for profiling is too limited. Since profiling codes are mixed in an OS kernel source code before it is compiled, the execution points that the developers can use for profiling is decided at compile time. Moreover, since the developers of the profilers cannot foresee all the possibility of profiling, this limitation sometimes limits what the developers can do.

In our study, we solved this trade-off problem by using richer symbol information, which is generated by a compiler extended by us. We proposed using DAOP for profiling an OS kernel because AOP is known as a good paradigm for logging. However, existing DAOP systems for the C language do not satisfy the both requirements: flexibility and abstraction. That is because

they can only use poor information. On the other hand, static AOP for the C language satisfy the both requirements though rebooting the OS kernel is required to change an aspect. The richer information narrows the gap between dynamic AOP and static AOP for the C language since disadvantage of existing dynamic AOP for the C language is that they cannot use source level information. Because of using poor symbol information, existing DAOP systems only provide limited pointcuts. They do not provide pointcuts to select data and a data flow. Providing those pointcuts is important for profiling the OS kernel because a member access of a structure is often used in an OS kernel, and data in the OS kernel is manipulated by multiple threads. Note that data manipulation by multiple threads cause tracing in naive ways, such as simple control flow tracing and data flow tracing, difficult.

From the top down viewpoint, our study provides profiling based on a data flow while existing profilers are based on a control flow (aka. cflow). Our prototype system provides flexibility in selection of a code for profiling and selection of execution points for profiling with good abstraction by AOP. It also helps the developers to selecting various execution points related to profiling that are difficult to select by hand.

From the bottom up viewpoint, our study provides good abstraction for a dynamic code instrumentation tool. Although those tools have flexibility, they only have assembly-level abstraction. Our study provides good abstraction for those tools by using richer symbol information.

1.4 The structure of this thesis

From the next chapter, this thesis presents technical background of this thesis, and details of the proposed mechanisms, including implementation to case studies. The structure of the rest of this thesis is organized as follows:

Chapter 2: Kernel Profiling

This chapter discusses required features for kernel profilers. We first discuss about the actual condition of profiling. Then we will figure out the requirements of profilers, and discuss for status of existing profilers.

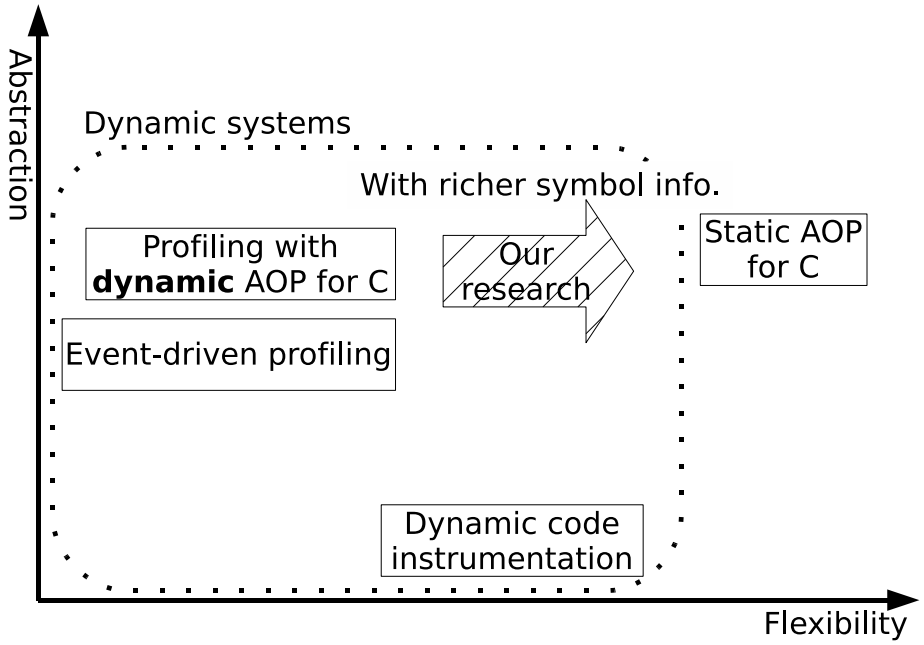

Figure 1.1. The position this thesis stands on

Chapter 3: Kernel Profiling by Dynamic AOP

To address the requirements shown in previous chapter, we propose using a dynamic AOP for profiling. We first explain what AOP is, then figure out existing AOP systems. Afterwards, we mention that existing DAOP systems lack pointcuts for selecting data.

Chapter 4: The Access Pointcut and KLASY

To fulfill a need for efficient pointcuts for profiling an OS kernel, we propose the access pointcut in this chapter. We explain design and implementation of the access pointcut. The implementation technique is named source-based binary-level dynamic weaving.

Chapter 5: The Xflow Pointcut and XenLASY

Then, to fulfill a need for efficient pointcuts for profiling an OS kernel, we also propose the xflow pointcut in this chapter. The xflow pointcut select an

execution point that data is in a specified data flow. We explain requirement, design and implementation of the xflow pointcut.

Chapter 6: Experiments

We did performance benchmarks for those pointcuts. We show results of those benchmarks. Then, to confirm availability on real applications, we also did case studies.

Chapter 7: Conclusion

Finally, we conclude this thesis in Chapter 7. We present contributions and future directions.

Chapter 2

Kernel Profiling

Operating system (OS) kernels are still under development even in 21 century. One of the big issues of development is performance improvement. For example, FreeBSD and Linux developers are still actively developing new schedulers [62, 51, 54, 78]. A new speed up idea proposed when new processor architecture proposed, e.g. symmetric multi-processor (SMP), virtual memory and so on. FreeBSD developers are implementing fine-grained lock for SMP systems even recently [42]. They are still discussing and developing technique to reduce system call overheads. Implementations of system calls are changed by the NetBSD and FreeBSD developers that received benchmark reports [76]. A new prefetching algorithm was recently proposed [28]. RedHat, Inc. proposed a tuned-up web server named TUX, which achieves its service inside a kernel module [60]. Thread implementation of Linux and FreeBSD are changed to use the native POSIX threading library (NPTL) and the kernel scheduler entities (KSE), respectively.

Good performance profiling tools are required for improving performance of OS kernels. Developers need appropriate tool sets for identifying performance bottlenecks and eliminating them. To identify performance bottlenecks, using a good performance profiling tool is mandatory. A naive way of performance investigation is manually modifying a kernel source code to insert measurement code. However, this approach needs recompiling and

rebooting, which can clear a memory image that caused performance degradation. This approach make the developers wait for a long time until some performance problem that they want to investigate occur again. Moreover, manually editing a source code is error-prone. The developers may forget removing a few code fragments and thereby cause a serious trouble, such as performance degradation and security information exposure. Even if they do not forget removing, there is also a pitfall. When they remove measurement code, they might wrongly remove statements irrelevant to the measurement. This can cause strange behavior that the developers hardly find the problem and the cause.

Moreover, virtualization technology improves importance of profilers. Virtualization technology is becoming popular recently for reducing the cost of equipment, the cost of electricity, the cost of machines and so on. It is also used to reduce space for machines. At this time, profiling becomes more and more difficult to do by hand because many subsystems are related to performance degradation. Data is manipulated not only each OS kernels, but also a virtual machine monitor. The Xen virtual machine monitor [8] is one of the famous virtual machine monitors (VMMs) now. In Xen, there are two kinds of virtual machines, Domain 0 and Domain U. Domain 0 is a privileged virtual machine that can manipulate native devices through device drivers. Domain U does not have a privilege. If Domain U should do some I/O, it requests Domain 0 to do the I/O instead of Domain U. The request will be send through pseudo-devices provided by Xen. Frankly, Domain 0 is proxy for Domain U. It is implemented with shared memory mechanism of Xen. In case of network I/O on Xen, a data of network I/O sent from Domain U is passed to Domain 0 through the Xen VMM, and is passed to a network device. As you can understand, the flow of data manipulation of network I/O is too long to understand it without support by tools.

2.1 Ideal profiling

The ideal way of profiling is inserting a profiling code snippet into arbitrary execution points related to what the developers want to investigate at run time without rebooting. For example, if developers want to investigate usage of inode, a code snippet like Figure 2.2 should be inserted to a kernel code like Figure 2.1 where the inode structure is used. This kind of insertion should be done at run time. In this example, value of the i_uid member of the inode variable is printed with a time stamp at the time when the i_uid member

```
int inode_change_ok(struct inode *inode, struct iattr *attr)
{
        int retval = -EPERM;
        unsigned int ia_valid = attr->ia_valid;

... snip ...

        /* Make sure a caller can chown. */
        if ((ia_valid & ATTR_UID) &&
            (current->fsuid != inode->i_uid ||
             attr->ia_uid != inode->i_uid) && ...
                goto error;

... omit the rest ...
```

Figure 2.1. Linux kernel source code (fs/attr.c)

```
struct timeval tv;
do_gettimeofday(&tv);
print_tv(tv);
printf("%ld", inode->i_uid);
```

Figure 2.2. An example of profiling code

of the inode variable is accessed. At this time, the execution points where profiling code snippets are inserted should be automatically selected by the name of some symbols of source-code level abstraction, such as a function name or a structure name.

2.2 Requirements for kernel profilers

For profiling appropriately, there are two important features required for kernel profilers. One is unlimited profiling points and code for profiling, and the other is abstraction of selecting an execution points for logging and writing a code. Profiling code should be able to be inserted arbitrary execution points where the developers want to insert code. If the developers want to investigate some performance bottlenecks, they will roughly investigate whole system first. They should measure the execution time of a large code section. Then they would gradually narrow the range of that code section for going into details to find accurate code snippets that cause performance degradation. At this time, they will measure elapsed time between given two

execution points. They must be able to give those execution points in kernel at runtime and change them, if necessary, without rebooting the kernel. Since rebooting the whole kernel is a time consuming task, frequent rebooting significantly decreases our productivity. Rebooting also clears the whole memory image and thus the internal data of the network module. After rebooting, the behavior that the developers want to investigate might disappear. Average time that the developers should wait for is $T : \int_0^T p(t)dt \geq \frac{1}{2}$, while $p(t)$ is a function that represents possibility of occurrence of the problem. It will be a long time if $p(t)$ is small for $\forall t$. Furthermore, the code snippet for measuring the elapsed time must be given by the developers since thy 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. In addition, the users may want to print a log message, for example, to record the value of an interesting variable.

Abstraction is an important feature for kernel profilers. Abstraction improves efficiency of kernel development. We mean higher abstraction for providing a source-code level view. The developers can easily do profiling if they can select an execution point to execute profiling code by a source-code level view or a syntax level view; function names, variable names and so on. Under this point of view, ability for selecting a member access of a structure for a profiling point is important. Polymorphism-like structures are used so many times in OS kernels. For implementing the polymorphism-like structures, a structure of the C language syntax is used. You can see this in file systems, device drivers and so on in an OS kernel. Moreover, structures are also often used to share a series of data in multiple modules instead of using naive arrays and variables. If the developers should specify a position to insert profiling code by memory address, profiling will become a hard work. A naive way of pointing the position is pointing some address of the instruction or code section that the developers want to investigate. However, pointing by hand is error-prone, and the developers may point out the wrong position. It is also annoying work to find the precise memory address by hand although they can know the address from debug information made by a compiler. The developers cannot effectively investigate bottlenecks without abstraction.

Abstraction also helps the developers to write code easily. Profiling code should also be written with high-level language on source-code level abstraction, such as the C language. If they can write profiling code with their favorite language, they can do profiling efficiently. Note that most of com-

modity OS kernels are written in the C language, they should be familiar with the C language. Obtaining a variable should be done with the C language abstraction instead of assembly level or machine level abstraction. A naive way of obtaining a variable is pointing some memory address or register name where these data are stored. However, if the developers should specify an instruction address to investigate, a memory address or a name of the register that has a variable to investigate manually by assigning the variable, profiling will become more difficult work to do. It is true that this information can be got from debug information made by a compiler. However, getting this information manually and writing a code to get values of some variables manually is too annoying to do.

2.3 Kernel profilers

The developers do not have to edit the source code of the kernel directly only for performance profiling if they use kernel profilers. However, existing kernel profilers are not sufficient to do ideal profiling shown above. There are two kinds of existing profilers, event-driven type and dynamic code instrumentation type. Both kinds of them do not fulfill the requirements shown above. We go into details why existing profilers are insufficient in following sections.

2.3.1 Event-driven type profilers

A typical event-driven type profiler such as LKST [33] can produce a trace log that records when a kernel event specified at compile time occurs. To use this kind of profilers, the users should use a patch distributed by a creator of each profiler. For applying the patch, a patch program automatically edits source code to produce a trace log at some execution points. Logging code will be executed at those execution points according to the users' specification given at run time.

Unfortunately, the flexibility of this approach is low; the users must statically determine the kernel events that can be recorded in a trace log when they occur. They cannot measure the execution time of code sections that were not modified before compile time. This is a serious problem because the code sections that they want to measure will change during a session of performance tuning. Moreover, since those profilers are not assumed to measure

some code sections by their creators, it is difficult to use for measurement of some code sections.

LKST

First we will take a look at LKST [33, 36]. LKST (Linux Kernel State Tracer) is a kernel profiler developed by researchers in Hitachi, Ltd. and Fujitsu, Ltd. LKST records trace logs at the execution points that is specified by the users. It can record various kinds of events; context switches, signals, memory allocation and so on. The users of LKST can dynamically change events to record. The users can avoid obtaining information that is not currently concerned by them. The users can reduce overheads of event tracing because they can turn off event loggers for unnecessary information. It is also possible for the users to change event handlers. They can use arbitrary code for event handlers by writing a program with the C language. The event handler created by the users is made as a kernel module, and registration of an event handler is done by loading it to kernel. Note that the default event handler for each event is just recording the events with some arguments. For example, pointers to the previous and next task_struct structures, process status, a process ID are recorded for a log at context switch.

Each log is read from a kernel buffer to a file by following command:

```
% lkstbuf read -f logfile
```

and displayed with command:

```
% lkstbuf print -f logfile
```

Then, following description is printed:

```
event_type=context_switch
        cpu=02, pid=00001008
        time=Tue Mar 26 18:48:53.143274134 2002
        arg1=0xd68a4000 0x00000000 : pointer to task_struct(prev)
        arg2=0xf2dde000 0x00000000 : pointer to task_struct(next)
        arg3=0x00000001 0x0000000a : process state/process count

event_type=mem_get_freepage
  ... snip ...
```

Enabling and disabling of event handlers are configured with a set of bit map masks, called *maskset*. If the users want to enable an event handler, they should turn on the bit in the maskset. The users should first extract maskset data to a file:

```
% lkstm read -m 0 -d | grep 0x > masksetfile
```

Here is an example of the masksetfile file:

```
  :              :
0x07f          0x01
0x080          0xff
  :              :
```

If the users want to record a log at the spin_lock function, they should change the bit of 0x080 to 0x01 (now it is 0xff). Note that the event number for the spin_lock function is 0x080. After the change of the bit, the maskset is loaded into the kernel:

```
% lkstm write -m 0 -f masksetfile
```

Here the users can use multiple-maskset and -m option of the command represents the ID number of the maskset. It is also possible to add an event to LKST by editing lkst_etypes.h file. Arbitrary execution points can be selected for getting a log. However, recompiling and rebooting the OS kernel is required for adding and removing the execution points for logging.

The fact that recompiling and rebooting the OS kernel are required for adding or removing the execution points to get a log is not an acceptable drawback. The users can hardly do ideal profiling shown in Section 2.1. If they try to do profiling shown in Section 2.1, member accesses should be the execution points. However, it is unrealistic. Generally, if the users want to insert a profiling code into arbitrary execution points that is related to the profiling, all the related execution points should be added before profiling. It is usually impossible because execution points that the users want to investigate will usually change during investigation since they will narrow the range of code section to measure an elapsed time. Approach adding execution points that supposed to be related to profiling do not work well. That is because the approach will cause unacceptable performance overhead of both kernel code size and execution time. Even if nop instruction is used at an inactivated execution point to reduce overhead, we believe that rate of cache

miss for the instruction cache will increase. Moreover good abstraction is required for selecting the events if the number of events becomes large. If LKST only provided abstraction as it is now with large number of events, selecting events would be more and more hard work to do.

LTT

LTT (Linux Trace Toolkit) [77, 56] is a tool for tracing events occurring in the Linux kernel. The users can examine event flow by per process basis with a graphical interface. The users can also get a text-formatted data for performing various calculations. With this tool, the users can know how each processor utilized, and how each process is scheduled in an OS kernel. LTT consists of a kernel patch, a kernel module, a trace daemon, and a data decoder. It also has a real time Linux support. The kernel patch specifies the execution points to get logs. The kernel module stores logs. The trace daemon read logs from the kernel buffer of the kernel module. The data decoder decodes logs into human readable format and performs analysis. To use LTT, the users should first execute a patch program to modify a kernel source code, and then recompile it and reboot the kernel. Then they should load the kernel module and execute the trace daemon.

Although the interface the users can investigate performance bottlenecks in per-process basis is useful, the execution points where the users can get a trace log is too limited. The users can hardly do ideal profiling shown in Section 2.1. Events they can know are entry and exit of system calls, traps, interrupts, and so on. It should be useful for profiling a process, but the users can hardly get a precise execution point of a performance bottleneck. That is because a series of execution points related to some data structure should be selected for doing this kind of profiling. However, this is difficult to do with this system. Moreover, the kernel with LTT always has some overheads because of its implementation. Logger cannot be disabled even at the execution points that are not related to the profiling. Increasing the execution points for profiling is impractical.

DTrace

DTrace [12] is an event tracer developed for the Solaris operating system. It is ported to some other open source operating systems such as FreeBSD. It is developed by researchers in Sun Microsystems, Inc. It is consist of *DTrace consumers* and *DTrace providers*. In the Solaris kernel, there are more than

30 thousands execution points available for executing profiling code. They are called *probe*s. A program that provides probes is called the DTrace provider. The users can select some of them to get information from the program. A program that gets information from the DTrace provider is called the DTrace consumer. If a probe is activated by using the DTrace consumer, the DTrace consumer will be called back from the DTrace provider when the CPU thread reaches the probe. Plainly speaking, DTrace is a feature for tracing behavior of a kernel and an application according to the command by the users. Note that each probe is well structured and the users can easily specify which probe to use.

The users of DTrace can execute an arbitrary code snippet at the probe. That is called the D script. Figure 2.3 shows syntax of the D script. The first line specifies a name of a probe. It is said as *probe description* in DTrace. The users can write more than one names here. The users can use wild card to specify the name. For example, if they want to select all system calls that are started from open, the name will be:

```
syscall::open*:entry
```

All probes will be selected if they do not write anything. If the users want to select all probes provided by the syscall provider, the name will be:

```
syscall:::
```

The second line represents a condition when the following script is executed. It is said as predicate in DTrace. For example, the users want to execute following script only when the thread of CPU 0 reaches to the probe, the condition will be:

```
cpu == 0
```

A block surrounded by braces ({}) shows a code snippet executed at the selected probe if the condition is fulfilled. It is said as *action* in DTrace. The code snippet is looks like a Perl or C program. The users can use arrays, functions, and variables in the action. Memory for arrays and variables are automatically allocated by DTrace. The users do not have to think about memory allocation. The users can enable the script with command:

```
# dtrace -s example.d
```

probe name, probe name, $\cdots$
/condition/
{
  A script to execute.
}

Figure 2.3. Syntax of DTrace code

```
syscall::read:entry
{
 self->t = timestamp;
}

syscall::read:return
/self->t/
{
 printf("%d%d spent %d nsecs in read\n",
  pid, tid, timestamp - self->t);
}
```

Figure 2.4. An example of DTrace code

It should be executed by the root user.

Figure 2.4 is an example of DTrace code. This example shows the elapsed time of the read system call of each process. In the example, current timestamp is stored to thread local storage (self->t) at the entry point of the read system call (syscall::read:entry). At the exit point of the read system call, difference of current time stamp and the stored timestamp (self->t) is printed with a process ID and a thread ID. To avoid executing the code of the exit point before storing timestamp at the entry point,

/self->t/

is used. This is the same as:

/self->t != 0/

Since all undefined variable is initialized with 0, /self->t/ is false if the code snippet for the entry point is not executed.

Some readers might think that ideal profiling shown in Section 2.1 can be done with similar way to Figure 2.4. However, it is not done with the straight way. To measure the elapsed time among execution points that is related to profiling target shown in the ideal profiling, the users should enumerate all related probe names. However, the number of execution points that the users should enumerate is too large to enumerate all the related execution points precisely. Even enumeration of some parts of them will take a long time. Moreover the users should have deep knowledge for the kernel for complete enumeration of related execution points because they should know which functions are related to the profiling target.

Another drawback of DTrace is that DTrace does not support Linux. Although some people are trying to port DTrace to Linux, it cannot be merged to Linux because of a license issue. Since DTrace is distributed under CDDL (Common Development and Distribution License), its license is not compatible with GPL (GNU General Public License) version 2, under which Linux is distributed. Even if DTrace is distributed under GPL version 3 as Jonathan Schwartz suggested, GPL version 3 is not also compatible GPL version 2.

SystemTAP

SystemTAP [59, 68] is tracer for Linux. It is first developed by researchers of RedHat, Inc., IBM, Inc., and Intel, Inc. The users can gather information about running Linux. The users can write some code snippets for profiling in simple scripting language similar syntax to the C language and Perl. It is called a *tapset*. They can insert the code snippets into the kernel from a command line interface.

Syntax of SystemTAP code is shown in Figure 2.5. *Probe point*s after the probe key word represent execution points at which the following script is executed. The users can specify events, functions, and so on. If the users want to select the execution point when the sys_open function is called, they should write:

```
kernel.function("sys_open")
```

They can also write:

```
timer.ms(200)
```

probe *probe point, probe point, ···*
{
  A script to execute.
}

Figure 2.5. Syntax of SystemTAP code

```
probe kernel.function ("vfs_writev")
{
    mode = $file->f_mode
     :
```

Figure 2.6. An example for getting a target variable

to execute the script in every 200 milliseconds. The users can use wild cards instead of enumerating function names. For example, if the users want to select all functions in the net/socket.c file, they will write:

kernel.function("*@net/socket.c")

The users can use special functions to get a current thread ID (tid()), a current process ID (pid()), a current program name (execname()), and so on. For accessing local context at the selected probe points, there is a mechanism called *target variables*. Figure 2.6 is an example using the target variable. In this program, a struct file * argument of the vfs_writev function can be used in the script. Note that the symbol for the variable should be remained to use this function. Some optimization can make those values into unreachable nonexistence.

The users can use variables, functions, arrays, and aggregates for writing the code snippets. Similar to DTrace, memory for variables, arrays, and aggregates are automatically allocated. Figure 2.7 is an example of a function. It will calculate factorial. A function will get any number of string or numeric arguments, and may return a single string or number. Note that types of parameters are inferred.

Doing ideal profiling shown in Section 2.1 is difficult. If the users try to do ideal profiling shown in Section 2.1, they will write code similar to Figure 2.8. The code first initialize before_time variable with -1. It is shown in probe begin. The users should enumerate functions that are related to profiling

```
function factorial (i)
{
    if (i < 0)
        return 0;
    else if (i == 0)
        return 1;
    else
        return i * factorial(i - 1);
}
```

Figure 2.7. An example of a function of SystemTAP code

target. The (A) line represents this. Then, following script prints elapsed time between functions that are related to the profiling. Here, enumeration of functions is difficult. Although the users can use wild cards for selecting functions to reduce difficulty of selection, the number of functions related to some processing is large. Moreover, the users should have precise and deep knowledge for the Linux kernel to selecting functions correctly. It is not realistic that the all users that profile the kernel have deep knowledge for it.

2.3.2 Dynamic code instrumentation type profilers

An on-line dynamic kernel instrumentation tool such as Kerninst [71] allows the users to modify a running OS kernel so that a given function will be called when the thread of control reaches the specified machine address. Since it directly replaces several machine instructions of the running kernel, the users do not have to reboot the kernel. This feature is significantly useful because unexpected performance behavior that the users want to investigate often occurs a long time after a kernel is booted. A reboot-less tool improves the productivity of kernel development.

However, since dynamic code instrumentation tools were not dedicated to kernel profiling, the abstraction provided by them is a quite low level. They provide only limited capability for kernel profiling. Thus, using those tools for performance profiling is a complicated task. Although the users can write profiling code, they should write code in the assembler-level abstraction.

```
global before_time

probe begin {
    before_time = -1
}

probe kernel.function ("..."), kernel.function ("..."),.... (A)
{
    after_time = gettimeofday_s()
    if (before_time != -1) {
        printf("elapsed time %s %d\n",
            probefunc(), after_time - before_time)
    }
    before_time = after_time
}
```

Figure 2.8. Doing ideal profiling with SystemTAP

Kerninst

Kerninst [71, 72, 74] is one of the dynamic code instrumentation tools for Linux and Solaris. It can dynamically add code into the running OS kernel anywhere anytime. Removal of code can be also done without rebooting the kernel. Figure 2.9 shows the overview of Kerninst. Kerninst consists of a kernel module and a userland daemon program, named kerninstd. The userland application will communicate with the daemon for kernel instrumentation. Note that no patches are required to use Kerninst.

There is an API provided by Kerninst to instrument the running OS kernel. The users will write code with the API for changing its behavior. The program written by the users using the API will communicate with kerninstd to send and receive command to the kernel module of Kerninst. Then the kernel module will modify the running kernel. Although the API is written in C++ language, the usage of the API is looks like an assembly-like language. The code the users should write to manage the running kernel is given in an abstract syntax tree (AST) style.

For example, the code for calculating '$a = (b + c) * d/(e - f)$' at the beginning of the inode_change_ok function is shown in Figure 2.10. In the code, there are three code sections, named (1), (2), and (3). The (1) code section will get references to the variables from their memory address and

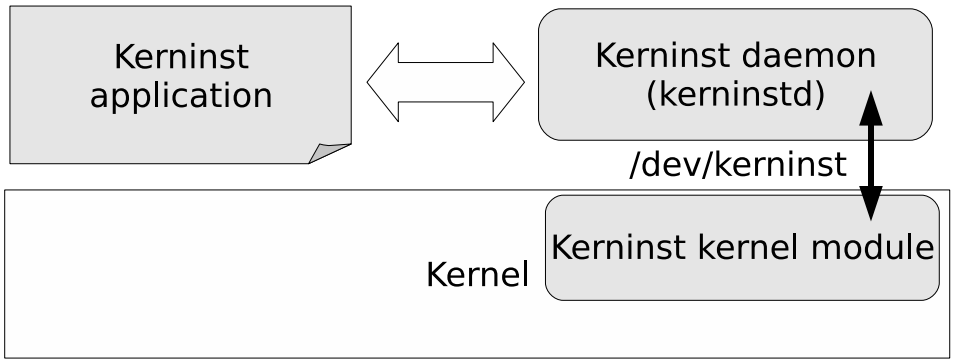

Figure 2.9. Kerninst overview

make them available in following code. kapi_int_variable is used to get an instance of an integer type variable whose address is given by its argument. Here addresses of a, b, c, d, e, and f are though to be known. The (2) code section will generate a code to calculate '$a = (b + c) * d/(e - f)$'. In the program, code for calculation is written one by one. It is hard work to do without making some mistakes. The (3) code section will insert code made at the (2) section to the beginning of the inode_change_ok function. Note that error handling is omitted in this code. Every Kerninst functions in the (3) section will return -1 if the operation failed. It first connects to the Kerninst daemon, then gets the entry point of the inode_change_ok function. After this, a code snippet made at the (2) code section is inserted to the kernel by the kmgr.insertSnippet function.

Since writing whole code in AST style is hard to do, there is an API to call a function in a kernel module. Of course, the function can be written with C language. It is easier to write code with Kerninst if the users call the function instead of writing whole code with the Kerninst API. Figure 2.11 and Figure 2.12 are example code using this feature. Suppose that the users want to print a log message with a time stamp when inode_change_ok() function is called. The log message includes the value of i_uid member of local variable inode declared in that function. They first define a function for printing a log message. The function is shown in Figure 2.11. It gets the frame pointer to investigate values of the local variable, inode, and print its member i_uid with a timestamp by using the printk function, which is the printf function used inside Linux kernel.

This function is compiled as a kernel module and dynamically loaded into the kernel address space. Note that the value of the local variable inode is obtained by ebp[11], where ebp holds the value of the ebp register when the

inode_change_ok() function is running. 11 is the offset of inode in the stack frame. It must be manually calculated from the memory layout of a stack frame. Then the users use Kerninst to insert a machine instruction at the beginning of the inode_change_ok() function so that this print_log() function will be invoked. The program is shown in Figure 2.12. It is compiled and run as a user process. It first connects to the Kerninst manager process and requests to modify the kernel code. Then it finds the addresses of the entry points of the two functions inode_change_ok() and print_log(). Finally, it inserts a code snippet "*hook*" so that inode_change_ok() will first call print_log().

The program shown in Figure 2.12 looks simple because Kerninst provides a mechanism for finding the address of the entry point of a function. However, if the users want to print a log message in the middle of that function to do ideal profiling shown in Section 2.1, the users must manually calculate the address where the *hook* code is inserted and then the users must pass that address to insertSnippet() instead of entries[0]. An example code for inserting a code into an arbitrary execution point is shown in Figure 2.13. As mentioned before, the users should calculate an address to insert a hook by themselves, and the address is given as the inst_ptr argument. Kerninst only helps us find the addresses of the entry and exit points of a function. It can also report the entry address of every basic block but the users have to identify which statements in a source file each basic block corresponds to by hand.

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

GILK

GILK [57] is a similar tool to Kerninst but it uses only the jmp instruction to insert hook code. The users can insert a code snippet before and after basic blocks. On the other hand, Kerninst will insert the break point trap (int3) if there is not enough space for a hook in the x86 architecture. This is achieved through an implementation of runtime code splicing. For inserting a code snippet, the users will first make a kernel module including a function to

Kinds of profiler	Flexibility	Abstraction
An event-driven type		✓
A dynamic code instrumentation type	✓	

Table 2.1. Comparison among Existing Systems

execute at a target execution point. Then, they will load the compiled kernel module, and insert a hook from the target execution point. While Kerninst provides the API that enables the users to write some code snippets with it, GILK do not provides this kind of API. The code snippets given by the users for profiling is written in the C language as a loadable kernel module. The performance of GILK is better than that of Kerninst but GILK supports only old Linux kernels.

The ideal profiling shown in Section 2.1 is difficult to do with GILK. The selection of execution points for executing code snippets given by the users is difficult to do because GILK does not provide any mechanism to make selection easy. Moreover, since GILK only insert code into the beginning and the ending of a basic block, the users cannot insert code snippet before a member of a structure is accessed. We cannot do ideal profiling with GILK.

2.4 Summary

In this chapter, we proposed requirements for kernel profilers. The requirements are unlimited profiling points and code for profiling, and abstraction of selecting an execution points for logging and writing a code. Those are required feature to do profiling shown in Section 2.1.

We also confirmed that no profilers and code instrumentation tools fulfill whole requirements we proposed. There are two kinds of kernel profilers, event-driven type and dynamic code instrumentation type. However, as shown in Table 2.1, neither kind of them fulfill our requirements. The former tools provides good abstraction but execution points that can be used for profiling is quite limited. The latter tools enable users to insert arbitrary code into an arbitrary position but their abstraction is bad. To do ideal profiling shown in Section 2.1, we need a tool that provides both source level abstraction and high flexibility.

```
kapi_manager kmgr;
void insert_calculation() {
    kapi_module kmod;
    kapi_function kfunc;
    kapi_vector<kapi_point> entries;

    /* (1) */
    kapi_int_variable a(address_of_a);
    kapi_int_variable b(address_of_b);
    kapi_int_variable c(address_of_c);
    kapi_int_variable d(address_of_d);
    kapi_int_variable e(address_of_e);
    kapi_int_variable f(address_of_f);

    /* (2) */
    // b + c
    kapi_arith_expr b_plus_c(kapi_plus, b, c);
    // (b + c) * d
    kapi_arith_expr bc_times_d(kapi_times, b_plus_c, d);
    // e - f
    kapi_arith_expr e_minus_f(kapi_minus, e, f);
    // (b + c) * d / (e - f)
    kapi_arith_expr result(kapi_divide, bc_times_d, e_minus_f);
    // a = (b + c) * d / (e - f)
    kapi_arith_expr code(kapi_assign, a, result);

    /* (3) */
    kmgr.attach("localhost", the listening port of kerninstd);
    kmgr.findModule("kernel", &kmod);
    kmod.findFunction("inode_change_ok, &kfunc);
    kfunc.findEntryPoint(&entries);
    kmgr.insertSnippet(code, entries[0]);
}
```

Figure 2.10. An example of Kerninst code

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

Figure 2.11. An example of a function to get a log by Kerninst code

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

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

Figure 2.12. A Kerninst code for calling a function from a hook

```
kapi_manager kmgr;
void insert_hook(kptr_t inst_ptr) {
    kapi_module kmod;
    kapi_point insert_point;
    kapi_function pfunc;
    kapi_vector<kapi_snippet> args;
    kapi_call_expr hook;

    kmgr.attach("localhost", 32770);
    kmgr.createInstPointAtAddr(inst_ptr, &insert_point);
    kmgr.findModule("profiler", &kmod);
    kmod.findFunction("print_log", &pfunc);
    hook = kapi_call_expr(pfunc.getEntryAddr(), args);
    kmgr.insertSnippet(hook, insert_point);
}
```

Figure 2.13. A code for calling a function from an arbitrary execution point

Chapter 3

Kernel Profiling by Dynamic AOP

We propose to use dynamic aspect-oriented programming (DAOP) for profiling an OS kernel. Since this is a dynamic system, profiling can be done without rebooting the OS kernel. This helps developers for profiling efficiently because rebooting the OS kernel flushes kernel memory whose status is difficult to occur again. Aspect-oriented programming (AOP) provides a higher-level abstraction. It enables the developers to select execution points to insert profiling code in source-code level view. The execution points the developers can select are function calls, function executions, assignment of variables, and so on. It is also widely known as an excellent paradigm for logging and profiling. If we use an AOP system, profiling code can be described as a module separated from the kernel source files. That separate module is called *aspects*. An aspect consists of *pointcuts* and *advices*. An advice is a language construct similar to a function. It is invoked when the thread of control reaches execution points specified by a pointcut. A pointcut is a composition of several predicates; it selects execution points that match those predicates. Those execution points are also called *join points*. To bind an advice with a target program at the execution points specified by a pointcut is called *weaving*, and unbinding the advice is called *unweaving*. It is normally part of a compilation process or a program-transformation process after compilation. We go into details for AOP in next section.

3.1 What is AOP?

AOP is a programming technique to solve a cross cutting concern. It helps developers to see programs with proper abstraction. It also helps them to write a program based on the abstraction. To let them write a code from proper abstraction, AOP provides a mechanism to write a series of code without considering about main logic. The code is called *aspect*. The developers can write an aspect that is related to various modules. For example, if they want to write logging code, a series of positions that developers want to get logs will be listed into one aspect code. It can also write more than one codes inside aspect. For example, developers want to write a disk cache mechanism, they should write code to get information of a disk and a virtual memory sub system at the same time. These codes will be written to one aspect. Other typical example for AOP is distributed computing, security, transaction, profiling, session management, and database management.

Similar to other separation techniques, a goal of AOP is improving modularity of design and implementation. In other words, its goal is localizing a concern into a module, and providing a good interface. If the goal is completed, developers will get benefits of modularity; modification can be done without thinking about other modules, development of each module can be done separately, and modules can be removed or changed to similar modules that have the same interface.

There are three important elements in AOP. They are join point model, *pointcut* and *advice*. *Pointcut* is a mechanism to select a join point, and *advice* is a series of code that affects at a join point. A join point is a moment at which a proper event occurs. It is a function call, a variable assignment, a control flow of a function, and so on. Some of join points, such as a function call and a variable assignment, correspond to execution points in the program. Those execution points are called join point shadows [47, 48]. Thinking about the AspectJ, which is the most popular AOP language for Java, there are various join points: calling and execution of each method, reading and writing of each member, call flow, and so on.

Pointcut

Pointcut will select a join point from the program. For example, following pointcut will pick out the call of setX method Point class whose return value is void:

```
call(void Point.setX)
```

As you see in the example, the call will pick out the call of some methods given as its argument. List of pointcuts supported by AspectJ is shown in Table 3.1. Here, We listed the pointcuts available for profiling. Pointcuts can be conjunct with && (and) and || (or). && is used to limit the join point that satisfy all pointcuts conjunct with &&. Besides, || is union of all pointcuts. There are also special meaning characters ! (not), *, .., and +. ! will exclude join points specified by following pointcuts. *, .. and + is used inside pointcuts such as call and execution. * means any characters except white space and . whose number is more than or equals to 0. .. means a conjunction of characters. + means inclusion of subclasses of classes and implementation of interfaces. For example,

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

will select the calls to both setX method and setY method of Point whose return value is void.

```
call(void Point.set*)
```

will select the all calls whose prefix is set in the Point class, and whose return value is void.

For ease of understanding, we also show some join point shadows in Figure 3.1. They correspond to some pointcuts in Figure 3.1.

Advice

Advice is a method-like code snippet executed at the join point selected by a pointcut. AspectJ has three kinds of advices: before, after, and around. Before advice runs before the join point selected by the pointcut. After advice runs after the join point selected by the pointcut. Around advice runs instead of the original code at the join point selected by the pointcut.

Narrowly defined meaning of aspect is a pair of pointcut and advice shown before. Figure 3.2 is an example of an AspectJ aspect. This aspect will select join points where void Point.setX(int) and void Point.setY(int) are called. These join points are named as move(). Before the call of methods specified by move(), a message "Entry of : <*method name*>" is printed through print method defined in the aspect.

Designator	Join points
call(*void Car.run(int)*)	a call to *void Car.run(int)*.
execution(*void Car.*(*)*) throws *FuelEmptyException*.	the execution of any methods of *Car* class that is declared to throw *FuelEmptyException*.
set(*int Car.fuel*)	when *int Car.fuel* is assigned.
get(*int Car.fuel*)	when *int Car.fuel* is read.
cflow(*call(void Car.run(int))*)	any join point in the control flow of each call to *void Car.run(int)*. This include call itself.
withincode(*void Car.run(int)*)	any join points that are defined in *void Car.run(int)* method.
target(*zoo.animals.elephant*)	any join points where the target object is instance of *zoo.animals.elephant*

Table 3.1. Pointcut designators available for profiling in AspectJ

3.1.1 An AOP system for the C language

There is an aspect-oriented system for the C language. Since AOP is firstly started for research of solving a cross cutting concern in an object-oriented language, aspect-oriented language for the C language is not as famous as that for Java. It is not mainstream either. However, Coady showed that AOP is also useful for developing an OS kernel written in the C language in 2001 [21, 20, 17] Since an OS kernel is multi-layered system and consists of many modules, there is also a cross cutting concern. She showed that development of a disk cache has crosscutting concern in it. That is because the developers should write both memory manipulation code and disk I/O code, and those code should cooperate each other. We will first take a look at static AOP systems, and then go into dynamic AOP systems. Static AOP systems are not available for ideal profiling because rebooting an OS kernel is required for changing the profiling code. It is not acceptable because it clears memory image of the running kernel.

AspectC

AspectC [40] is an early static aspect-oriented language for the C language and it has been used for showing that aspect-oriented programming works well for modularizing an OS kernel [21, 18, 20, 17, 19]. Syntax of AspectC is

```
class Car {
    int fuel;
       :
    // Join point shadow: execution(void Car.run(int))
    void run(int distance) throws FuelEmptyException {
        // Join point shadow: set(int Car.fuel) and get(int Car.fuel)
        fuel = fuel - distance;
        if (fuel < 0)
            throw new FuelEmptyException();
        // Join point shadow: call(Car.drawCar())
        this.drawCar();
        return distance;
    }
}
```

Figure 3.1. An example of join point shadows

a subset of AspectJ. Since the C language is not one of the object-oriented languages, AspectC does not support object-orientation and clear modules. Instead of supporting these features, AspectC will limit extent of the weaving by file names and directory names. One of the famous examples for AspectC is writing a prefetch code by aspect. The mechanism for prefetching a disk block must cut across a virtual memory sub-system and a disk sub-system. Without AOP, developers should write a code into two sub-systems, which cause source code tangled and difficult to understand. Aspect will clear the behavior of prefetch code for both a virtual memory sub-system and a disk

```
aspect SimpleLogging {
  pointcut move():
    call(void Point.setX(int)) ||
      call(void Point.setY(int));
  before(): move() {
    print("Entry of : " + thisJoinPoint);
  }
  void print(String message) {
    System.out.println(message);
  }
}
```

Figure 3.2. An example of aspect of AspectJ

sub-system. The developers can see a series of prefetch code with reading only a file.

AspectC++

AspectC++ [69, 6, 45] is another static aspect-oriented system for the C/C++ language. It provides the same kinds of pointcut designators that AspectJ does [41] since it is a source-to-source translator from an aspect-oriented language to the regular C/C++ language. There is also an example using AOP for the eCos kernel [43]. eCos is one of OS kernels whose target is an embedded system. Tracing, interrupt synchronization, and event handling are separated by AspectC++ for solving cross cutting concerns. According to this study, performance of AOP is as good as a hand-written version.

3.2 Dynamic AOP

While applying aspects with static AOP systems require rebooting the OS kernel, dynamic AOP systems do not. They can weave and unweave the aspects without rebooting the OS kernel. They help the developers to do profiling efficiently. Since rebooting the OS kernel clears memory that may contain trails of some disordered behaviors, the developers should want to avoid it. Otherwise, they should wait for a long time until the behaviors occur again after weaving aspects.

Dynamic AOP systems also provide source-level abstraction as static AOP systems do. They enable the developers to efficiently select execution points to investigate. They also enable the developers to write code for profiling with a high-level programming language.

A large drawback in dynamic AOP systems is that they cannot use as rich information as static AOP systems can use. Since most of source-level information is discarded after compilation, existing dynamic AOP systems cannot provide pointcuts as various as static AOP systems do. Besides, since static AOP systems can utilize the complete source-level information of a program when weaving an aspect at compile time, they can easily provide various pointcut designators.

There are two kinds of techniques for implementing dynamic AOP system. One is dynamic code instrumentation, and the other is a virtual machine extension. The former kind of systems are PROSE [58] and Wool [63], and the latter kind of system is Steamloom [10, 34].

PROSE 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. The cost of handling breakpoint traps in the OS kernel is relatively smaller than in the JVM.

Wool is another dynamic AOP system for Java that does dynamic code instrumentation. 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 reloads 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.

Steamloom is a virtual machine (VM) for providing DAOP for Java. It is the first VM implementation of DAOP. Dynamic weaving is implemented as dynamic modification and reinstallation of method byte codes. Its implementation is similar to JVM HotSwap and SLIC [29], which is a dynamic extending system for commodity operating systems. However, it can achieve an aspect management. It is based on jikes Research Virtual Machine (RVM), research VM developed by IBM. Steamloom is lower overheads than the other DAOP systems that perform weaving by dynamic code instrumentation. Since recompilation of methods is done with full optimization, an application on Steamloom can run as fast as that woven with a static AOP system, such as AspectJ.

Existing DAOP systems for the C language are not sufficient for ideal profiling shown in Section 2.1 because their pointcuts are too limited. We need a DAOP system for the C language with pointcuts of a DAOP system for Java. An existing DAOP system for C language does not have pointcuts for selecting data well because details of data access are eliminated at compile time. Symbol information made by a C compiler is much less than that made by a Java compiler. It does not have detailed information of variables. Pointcuts for selecting data is required because an OS kernel has special characteristics compared to other small C applications. The developers should also use these characteristics for ease of development. We will first explain the characteristics, and then explain why existing DAOP systems for C is not sufficient in details.

3.2.1 Characteristics of an OS kernel

Characteristics of source code of an OS kernel are high-level modularization and multi-threaded data manipulation. To realize high-level modularization, a structure is shared among plural modules in a large-scaled C program such as an OS kernel. The structure contains a common data among modules related to each other. It is often passed through a function to another for sending and receiving data. It is also used to implement polymorphism-like mechanism in the OS kernel. This implementation is used in a device driver and a virtual file system of the kernel such as FreeBSD and Linux. There is strict coding convention for writing code. Since usage of each structure is ruled by the convention, the code using data structure has a pattern; it is only used among related modules.

Multi-threaded data manipulation is normally used in the OS kernel for coping with both low response time and manipulation of complex data. Programs that manipulate data in the kernel is separated into two threads, the top half and the bottom half. The top half manipulates complex data for providing higher-level abstraction to the userland programs. For example, main logics of a file system, a socket I/O subsystem, and a memory subsystem are implemented in the top half. The top half communicates with the bottom half through a queue and a trap. The top half appends data to a queue for sending it, and removes data from a queue for receiving it. Appending and removing are signaled through the trap. On the other hand, the bottom half manipulates a real device. Since response time and elapsed time by the bottom half should be low and small, respectively, its jobs are only simple things. Jobs for the bottom half are sending data in the queue to a device, and receiving data from a device to append it to the queue. Data arrival is signaled through the trap.

Since the OS kernel is multi-threaded application, profiling by tracing a call flow or a control flow is difficult. The call flow will not continue through a thread to another. Each data will be appended to a queue and the developers tracing through call flow cannot investigate how the data is manipulated after it is appended to the queue.

For tracing this kind of data, pointcuts that select data and a data flow are required. It should also support characteristics of an OS kernel; a structure is shared among modules related to each other. As we mentioned before, a structure is often used in a large-scaled C program. It is transferred among modules for sending and receiving a series of data. It is also used to implement polymorphism-like structure in the C language. If the developers use

the pointcuts, they can efficiently select modules that are using the data. This is much easier than using a pointcut that select a function such as the execution pointcut and the call pointcut. If the developers use this kind of pointcuts, they should have deep knowledge of an OS kernel for enumerating all the functions related to profiling. Otherwise, there is possibility for them to miss some important data. Moreover, the OS kernel should not be optimized since symbol information is eliminated by an optimization. If a function is inlined, the information for it will be eliminated. However, if the developers use the kernel that is not optimized for profiling, the result should become unrealistic. That is because an OS kernel is usually compiled with some optimization option such as -Os, -O1 or -O2, which inlines functions.

3.2.2 Existing DAOP systems for the C language

There are several dynamic aspect-oriented systems for the C language. Each system can weave an aspect into a running program written in the C language without recompiling and rebooting it. A target of some systems is a userland application, while a target of the others is an OS kernel. However, both kinds of existing DAOP systems for the C language are insufficient for profiling because they lack pointcuts for selecting data and a data flow. Here, we will take a look at them one by one.

DAO C++

DAO C++ [1, 2] is a dynamic aspect-oriented system for user processes written in the C++ language. For weaving an aspect, this system modifies the compiled binary of C++ programs during runtime. It is designed and implemented network application in mind, and it can weave and unweave an aspect through a network. For implementing DAOP, it first preprocess C++ program to make meta-object data and to insert hooks. Then, AOP engine works at runtime for dynamic weaving. Only the pointcut supported by the system is selection of a method call. That is because its target application is a dynamic adaptation. It lacks pointcuts for selecting data and a data flow.

TinyC^2

TinyC^2 [82] is a dynamic aspect-oriented system for userland application written in the C language. It experiments the idea of implementing an aspect-oriented language based on existing code instrumentation tool. In

its prototype, Dyninst [11] is used. Its AOP language is an extension of the C language. It provides the same mechanism with the before and after advices of the execution pointcut in AspectJ, although its syntax is different. It use the onentry and onexit designators as the before and after advices of the execution pointcut. The programs written by the developers will be translated into a code snippet using Dyninst API. The translated code is executed for dynamically weaving at runtime. Since its implementation is limited inside Dyninst and use the symbol information produced by a normal compiler, it does not have a pointcut that select data or a data flow.

Arachne

Arachne [22] is also a dynamic aspect-oriented system for user processes written in the C language. Although it also uses the symbol information produced by a normal compiler, the pointcut designators of Arachne cover not only function calls but also accesses to global variables and memory blocks allocated by the malloc function. However, pointcutting accesses to memory blocks imply serious performance penalties since Arachne uses a page fault for detecting accesses to the memory block. Arachne also showed that it is useful for changing behavior of Squid web cache without recompiling and rebooting it [65]. In [65], it is shown that Arachne is useful for security fix, changing prefetching mechanism, and adding ICAP support at runtime. Although Arachne has many pointcuts, it lacks a pointcut for selecting a member access of a structure at runtime. As shown before, the pointcut is required for efficient profiling of an OS kernel. Moreover, since a target of Arachne is userland application, it cannot be used for an OS kernel.

μDyner

μDyner [67] is a predecessor of Arachne. It is also a dynamic aspect-oriented system for user processes written in the C language. μDyner inserts hook code at every join point marked as *hookable* when a source file is compiled. The hook code examines whether or not the join point is selected by a pointcut during runtime and, if it is selected, the hook code executes the associated advice. Although μDyner potentially can support various kinds of join points since it inserts hook code at compile time, the developers must annotate source files by the hookable mark so that the hook code will be inserted at appropriate join points. If a join point is not marked as *hookable*, it cannot be selected by a pointcut during runtime. Furthermore, overheads

due to hook code are not negligible if the number of hookable join points is large.

Inserting hooks by AspectC++

[64] and [30] used AspectC++ for dynamic weaving. Their target applications are implementing software product line, the way of development to make software as if making a car. The users can freely choose parts for making their own software. Since hooks for dynamic weaving are inserted by AspectC++, code insertion to join points becomes easy task. However, it also has the same problem as μDyner has. Overheads of hook code should become negligible if the number of hooks prepared for join points is large.

TOSKANA

TOSKANA [25, 27] is a dynamic aspect-oriented system for the NetBSD operating system kernel. It dynamically modifies the compiled binary of the kernel for weaving an aspect. Its target application is autonomic computing. This system only provides the before, after, and around advice for the execution pointcut of functions. In its application examples, it is shown that TOSKANA is useful for adding self-configuration, self-healing, self-optimization, and self-protection features at run time. At the weaving, a kernel module is automatically replaced to adapt the kernel dynamically. Since its only pointcut is the execution pointcut, it is not useful for efficient profiling of an OS kernel.

TOSKANA-VM

TOSKANA-VM [26] is a system that allows developers to dynamically weave an aspect with the kernel. The approach of TOSKANA-VM is similar to the approach of Steamloom [10, 34], which is a custom Java virtual machine extended for enabling dynamic weaving. The kernel of TOSKANA-VM is compiled by a special compiler into virtual machine code, which is run on a virtual machine named LLVM. Since the virtual machine code contains rich symbol information, TOSKANA-VM allows developers to pointcut various kinds of join points such as reading and writing a variable. However, the kernel must run on a virtual machine and thus this approach cannot be used for profiling a kernel directly running on native hardware.

3.3 Summary

In this section, we proposed that dynamic aspect-oriented programming (DAOP) is useful for profiling OS kernels. Since aspect-oriented programming (AOP) first started from an object-oriented language, AOP for the C language is not as famous as that for Java. However, it is known that AOP is also useful for kernel development. For example, it is useful for implementing prefetch algorithm in an OS kernel. We first explained AOP for Java and mentioned static AOP for the C language. After that, we go into DAOP.

For DAOP, we first explained existing AOP for Java, and mentioned that some pointcuts of DAOP for Java are required for DAOP for the C language to do better profiling. We first mentioned characteristics of an OS kernel, and explained that existing DAOP systems for the C language lack pointcuts for utilizing the characteristics. The characteristics are that a structure is often used in an OS kernel, and that an OS kernel is a multi-threaded application. Since an OS kernel is a large program, a structure is used for providing good modularization. To cope with both lower latency and complex data manipulation, an OS kernel should be multi-threaded.

For utilizing this feature, pointcuts that select data and a data flow are required. However, existing DAOP systems for the C language do not have enough support for this feature. We showed several DAOP systems for C language and showed that all of them do not have support for the pointcuts.

Chapter 4

The Access Pointcut and KLASY

This chapter presents KLASY [80, 81], which is a dynamic AOP system for profiling an OS kernel. The previous implementation of dynamic AOP systems for the C languages lacks a pointcut to select a member access of a structure. Since a member access of a structure is often used among functions that related each other, the pointcut will help the developers to do profiling efficiently. The pointcut is called the access pointcut. This chapter presents usage of the access pointcut and our new dynamic weaving technique for implementing the pointcut.

The name of our new technique is source-based binary-level dynamic weaving. It collects richer symbol information including a position of a member access at compile time. The richer symbol information is used at weave time to get a memory address of a member access. At weave time, Kerninst [71] is used to insert a hook. Note that a hook is a code for calling an advice from arbitrary execution point in the kernel.

KLASY has a mechanism to get local variables for using inside an advice. With this feature, the developers can print values of those variables and change code behavior according to the values. The richer symbol information is also used to implement this feature. The memory addresses or the register numbers for the variables are recorded inside it. The unique feature of KLASY is that it can pass the instance of the structure whose member is

selected by the access pointcut. We modified the parser of the compiler to implement the feature.

The rest of this chapter is organized as follows. Section 4.1 explains necessity of the access pointcut in details. Section 4.2 shows the syntax and usage of our aspect language. Section 4.3 presents how the access pointcut is implemented. We conclude this chapter in Section 4.4.

4.1 Necessity of the access pointcut

Although AOP is known as a good paradigm for logging, existing AOP systems are not satisfactory for profiling the OS kernel. First, an AOP system we need is a dynamic AOP system, which can dynamically weave an aspect. As Kerninst does, the AOP system must be able to attach and detach a profiling aspect to a running kernel without rebooting it. There are several dynamic AOP systems for the C language but they support only limited kinds of join points. They do not provide significantly better functionality than Kerninst. For example, TOSKANA [25] allows users to select only function execution as join points. Arachne [22] allows users to select not only function calls but also accesses to global variables and arbitrary memory blocks. However, any of them does not enable selecting member accesses to structures.

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

Another limitation of existing dynamic AOP systems for the C language is a mechanism for context exposure. AOP systems have a mechanism for passing context information of a join point to an advice. For example, if

a join point is function execution, function arguments can be passed to an advice. Arachne [22] allows an advice to access a return value and global variables as well. However, existing AOP systems for the C language do not allow an advice to local variables visible at a join point. Exposing local variables at a join point to an advice might not be an appropriate design with respect to modularization but it is often necessary for profiling.

4.2 KLASY

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

An aspect for KLASY is written in C although it includes XML-like tags. Figure 4.1 is an example program. It can be compiled and woven by the klasy command at any time during runtime. For example,

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

weaves the aspect in the file inode_trace.klasy into the OS kernel. After this aspect is woven, it prints a log message when the i_uid member of the inode structure is accessed within the function body of inode_change_ok(). Note that the printk function is used as the printf function in Linux kernel, and the do_gettimeofday function is used to get a current timestamp. Unlike the example in Figure 2.11 and Figure 2.12, this aspect prints a log message not when the execution of inode_change_ok() starts but whenever the i_uid member is accessed in inode_change_ok().

An aspect is surrounded by the aspect tag. The import tag specifies the header files that are necessary to compile an advice body. When an advice body is compiled, two header files linux/kernel.h and linux/module.h are implicitly included. Other header files can be included by using the import tag. They are just translated into #include <*import content*> in the aspects body to be compiled. For example, the developers that want to use do_gettimeofday function, which is used for getting a current precious time,

they should write:

```
<import> linux/time.h </import>
```

If they want to know a page size (PAGE_SIZE) or the kernel page offset (PAGE_OFFSET), they should write:

```
<import> asm/page.h </import>
```

Developers can use their own header file by putting it to under the include directory of Linux kernel. For example, if they made their own header file with the name "local.h" and directly put it to the include directory, import description will be:

```
<import> local.h </import>
```

Any code can be written inside their own header file. It will be put at the top of the aspects to be compiled and loaded to the kernel later.

The advice body is a code fragment written in the C language and it is surrounded by the before (or after) tag. KLASY does not support an around advice but it is not a serious drawback since an aspect in KLASY is for profiling and it rarely needs around advice, which is mainly used to execute a function at a join point only if necessary. In an advice body, several characters must be escaped since an aspect is tagged in XML. For example, angle brackets (< and >) must be replaced with < and >, respectively. The ampersand & must be &. In an advice body, a special variable pc is available. It represents the current value of the program counter, that is, the memory address of the machine instruction selected by a pointcut.

The advice body follows a pointcut definition surrounded by the pointcut tag. KLASY currently provides seven pointcut designators: execution, access, within_file, within_function, target, local_var, and argument. The execution pointcut identifies function executions as join points. The access pointcut identifies both read and write member accesses to structures as join points. For example, if the developers write access(sk_buff.data), both read and write access to data member of sk_buff structure will be selected as join points. Therefore, advice code will be executed before (with before advice) or after (with after advice) in following situations:

```
skb->data = data;
```

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

Figure 4.1. An aspect written in KLASY (inode_trace.klasy)

and

```
unsigned long offset = new->data - old->data;
```

Wild cards (%) are available in these pointcuts. Since * represents a pointer type in the C language, wild cards are not * but %. Note that wild cards matches any characters except '.' (delimiter of a structure name and a member name) whose length is more than or equals to 0. For example,

```
skb.%
```

matches the any members of the structure whose name is skb.

```
skb.d%
```

matches the member of the structure whose name is skb and started from d.

```
skb.data%
```

matches the member of the structure whose name is skb and started from data. skb.data is accepted by this pattern.

%.data

matches the data member of any structures.

%.%

matches the any members of any structures. Since we use a regular expression library for matching in the background, our system potentially support the regular expression for matching. However, to make syntax of pointcuts similar to AspectJ, we do not use the regular expression itself.

If a join point selected by a pointcut is a member access, the target pointcut is available. It sets a given variable to a pointer to the structure that is accessed at the join point. In Figure 4.1, the pointer to the inode structure is bound to a variable inode_value that is available in an advice body. Even if a local variable is not a reference to a structure instance, KLASY will automatically calculate the reference from the local variable. For example, if the pointcut is access(file.f_owner) and sk->sk_socket->file->f_owner matched, sk->sk_socket->file will be automatically calculated from sk and the reference will be passed. Note that header files related to this calculation should be included by import tag. The type of inode_value is void*. If the developers want to access each value, they should first cast to the proper type they want to use. If the developers use target value pointcutted by access(sk_buff.data), they should cast the value to struct sk_buff* before using its value.

If the within_function pointcut is used, the local_var pointcut is used for obtaining the reference of a local variable at a join point. Syntax of local_var pointcut is:

local_var(*source local variable, destination advice variable*)

Since the reference of the local variable is passed as void* variable, the developers should cast it before using it. For example, if the developers want to use the local variable n inside advice and its name is np, they will write:

local_var(n, np)

To use the variable, they should also write following kind of a program in advice body:

```
... no 'j' is used ...// (A)
for (i = 0; i < 10; i++) {
  int j;
  ... // (B)
}
```

Figure 4.2. Special meaning of the local_var pointcut

```
struct sk_buff *skb = np;
```

The local_var pointcut can also be used to limit the range of pointcuts inside the scope of the local variable given by 1st argument of the local_var pointcut. For example, if the developers use the pointcut like in Figure 4.2:

```
local_var(j, j_pointer)
```

only join points in Section (B) will be selected and join points in Section (A) will not be selected.

The within_file and within_function pointcuts select join points included in a specified file and function body, respectively. If the developers want to limit join points inside net/core/dev.c file, they will write:

```
within_file(net/core/dev.c)
```

Similar to within_file, limiting join points inside skb_clone function will be:

```
within_function(skb_clone)
```

Developers can use wild card (%) in both the within_file pointcut and the within_function pointcut. For example, if the developers want to limit join points inside .c files that is started from ip_ in net/ipv4 directory, they will write:

```
within_file(net/ipv4/ip_%.c)
```

If the developers want to limit join points inside functions started from skb_, they will write:

```
within_function(skb_%)
```

Since within_file pointcut and within_function pointcut do not use complete matching, they will match strings that contain patterns. For example, if the developers write:

```
within_file(dev.c)
```

then, net/core/dev.c file is matched by this pattern. Because of the implementation, a function name that is matched by within_function pointcut should be a name of an exported function. To get a function name of join points, KLASY will resolve the name from symbol information by using Kerninst. At this time only a function name got by symbol information is a name of an exported function. Note that the within_function pointcut and the within_file pointcut cannot be combined with the execution pointcut. If they are combined, KLASY cause an error. For example, following code will cause an error:

```
execution(skb_clone) AND within_file(dev.c)
```

The argument pointcut is used for obtaining the argument value of a function specified with the execution pointcut. Syntax of argument pointcut is the same as that of local_var pointcut. Its syntax is:

argument(*source parameter*, *destination advice variable*)

For example, to get size parameter of alloc_skb function, the developers will write:

```
argument(size, sizep)
```

to get the reference to size parameter as its name is sizep variable. As the same as local_var pointcut, the reference of the argument is passed as void* variable. Developers should cast it to a proper type before using it. This pointcut can be also used to limit join points that have a parameter variable specified by the argument pointcut. Developers can select some functions with wild card and extract it with argument pointcut that has the parameter specified by the argument pointcut.

Multiple pointcuts shown above can be composed by AND or OR operators. If within_function is composed by AND operator with another pointcut such as access, the selected join points are ones that satisfy the conditions specified by both within_function and access. Developers can also use a parenthesis ('(' and ')') to specify order of pointcuts. Note that AND operator is evaluated before OR operator if pointcuts that contain OR operator is not in parentheses. For example, "A AND B" is evaluated before "C" in:

A AND B OR C

Pointcut expressions that are combined with AND operator must have more than one access pointcut or execution pointcut. Since this reason, following pointcuts are invalid:

within_file(sk_buff.c) AND within_function(skb_clone)

that do not have either access pointcut or execution pointcut. Even at the evaluation time, any pointcuts except access pointcut and execution pointcut should be combined with access pointcut or execution pointcut. Since this reason, following pointcuts are also invalid:

access(sk_buff.data) AND within_file(sk_buff.c) OR within_file(dev.c)

because within_file(dev.c) is not combined with either access pointcut or execution pointcut. After access(sk_buff.data) AND within_file(sk_buff.c) will be first evaluated, within_file(dev.c) will be evaluated. However, within_file(dev.c) is not combined with access pointcut or execution pointcut. Note that more than one access pointcuts are combined with AND operator for the purpose of pattern matching. For example, if the developers write:

access(sk_buff.%) AND access(%.data)

then, join points selected by access(sk_buff.data) will be selected. Developers can write similar thing with the execution pointcut. Note that the access pointcut and the execution pointcut cannot be combined with AND operator. That is because there is no join point selected by conjunction of those two pointcuts. For example,

access(sk_buff.data) AND execution(skb_clone)

will cause an error.

4.3 Source-based binary-level dynamic weaving

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

4.3.1 Extended symbol information

To enable the access pointcut shown above, the target OS kernel must be compiled by our extended C compiler so that the compiled binary will include the symbol information that is necessary to locate all occurrences of member accesses to structures. We extended the GNU C compiler (gcc) to develop that C compiler.

To record the locations of member accesses, we extended the parser of gcc. Since the global variables lineno and input_filename represent the current line number and the file name during parsing, our extended parser records the values of those global variables as well as a member name and a structure name whenever it encounters member accesses. Note that an abstract syntax tree produced by the gcc parser does not include type names. All type names are converted into integer identifiers. Hence our extended parser maintains a mapping from the integer identifiers to the type names. Our parser remembers the type names of structures and their integer identifiers at declaration time, and use them at use time. We use a hash table to remember them for performance improvement of compiling.

To locate the memory addresses of the machine instructions corresponding to the join point (shadow [47, 48]) of member accesses, KLASY also needs to know where the compiled code of a given source line is placed in memory. Although the original gcc produces such address information if it runs with a debug option -g, the produced information is not sufficient for KLASY. For example, consider pointcut accesses to the addr_limit member

of the thread_info structure. A source file acct.c of the Linux 2.6.10 kernel includes an access to that member at line 493:

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

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

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

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

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

with both the line 493 in acct.c and the line 32 in uaccess.h. These two line numbers associated with the code is removed by the RTL (register transfer language) generator and only the line 32 in uaccess.h is associated after that. We modified the RTL generator and the sub systems following the RTL generator, such as an RTL optimizer, and an assembler (gas), so that they can maintain multiple line numbers. In both systems, gcc and gas, multiple line information of line numbers are reduced to one by default; only the top of that information is remained and the others are eliminated. We changed optimization code of gcc and code of debug information generator of gas to have more than one mapping between lines and memory address of the lines.

Some readers might think that such complex implementation is unnecessary if all the source files of the Linux kernel are preprocessed in advance by the cpp command with the -P option. This option suppresses gener-

ating #line directives, which represent line numbers before preprocessing. However, this approach loses the information of the original line numbers and thus KLASY could not generate a helpful warning message including a line number. Furthermore, the users would want to know the original line numbers of the selected join points. Therefore, KLASY must maintain the original line numbers by the approach described above. Another problem of the cpp -P approach is that it cannot maintain correct line numbers if two source lines are merged into one for optimization.

Also, KLASY can maintain multiple line numbers for inline functions. Debug information has both information of a line number of an original program and a line number of a header file that has the inline function at the line. Although information for both an original program and the inline function is stored in debug information of the kernel, debug information of a header file will not used not to use join points twice.

To cope with a member access of a structure inside a parenthesis (or surrounded by '(' and ')'), we also modified the compiler. Problem cases occur with the if, while, and for statement. Because of gcc limitation, a mapping between a line number and a memory address is not made for a statement in parentheses. For example, if the program is:

```
629: if (skb_copy_bits(skb, -head_copy_len, head + head_copy_off,
630:                   skb->len + head_copy_len))
```

then, only a mapping for the line 629 and the memory address for the line is made while a mapping for the line 630 and its address is not be made. If the developers write pointcut access(skb.len), skb->len in line 630 should be selected. However, its address cannot be resolved since the mapping for the line do not exist. Here we use the mapping for line 629 instead of the mapping for line 630 since they are not far and they are in the same basic block. To implement this feature, our compiler remembers the line number where the statements that has '(' parenthesis exist, and uses its line number instead of line numbers of statements in parentheses.

4.3.2 Dynamic weaving

An aspect is compiled by the aspect compiler of KLASY. The advice bodies in the aspect are compiled into a loadable kernel module. It is loaded in the kernel space by the insmod command of Linux. Then KLASY resolves the memory addresses of the join points selected by pointcuts and KLASY inserts *hook* code at those addresses by using Kerninst [71]. The hook code calls an

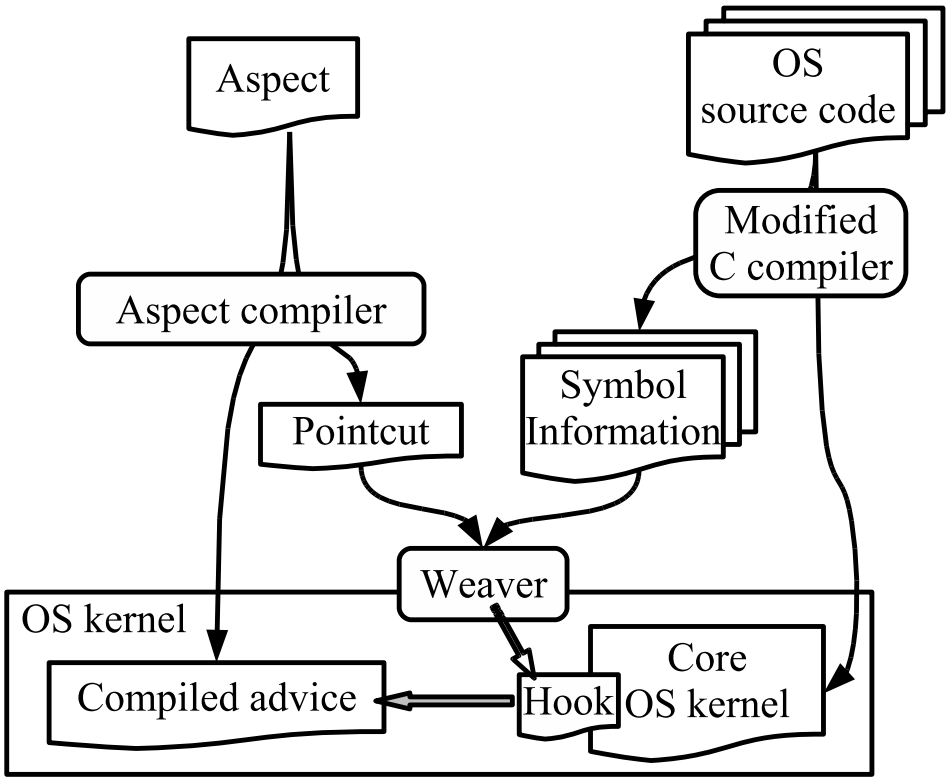

Figure 4.3. Overview of KLASY

advice body when it is executed. The overall architecture is illustrated in Figure 4.3.

KLASY resolves the memory address of a join point by analyzing the symbol information produced by our extended gcc compiler. It first searches the symbol information for the file name and the line number of each join point selected by a given pointcut. Then it resolves the memory address corresponding to that line number. Our extended compiler compiles the OS kernel with the -g option and stores the extended address information in the debug information section of the compiled binary code. The compiler uses the DWARF 2 format [73] to construct the binary code. KLASY first reads the .debug_info section of the binary of the kernel and finds the address information of the compilation unit that corresponds to the file name. Since the address information consists of .debug_line sections, KLASY reads them to find the memory address that corresponds to the line number. In KLASY, this resolution will be done by two daemons, symmapd and linemapd.

Symmapd resolves line numbers and file names from a member access specified by the access pointcut. Linemapd analyzes debug information written in DWARF 2 format, and resolves a memory address from a line number and a file name. Note that mappings between member accesses and line numbers are calculated and stored in a hash table before dynamic weaving starts for performance improvement. Without calculating before weaving, weaving will be too slow to use.

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

Since the compiler may merge multiple lines for optimization, some lines including selected join points may disappear from the compiled binary. KLASY cannot find the memory address of those lines. However, in that case, KLASY tries to find the memory address of the line nearest to the original line. KLASY increments and decrements the line number one by one and tries to find the memory address of the line. When KLASY finds the memory addresses of the two lines before and after the original line, it examines whether or not the two lines are in the same basic block. If they are, KLASY adopts the memory address of the line before (or after) the original line for a before (or after) advice. We assume that, if the two lines are in the same basic block, the original line is also in the same block and the original line is surely executed between them. Otherwise, KLASY prints a warning message and ignores the join points included in the original line. The basic blocks are computed by Kerninst.

If the memory address of a join point is found, KLASY inserts hook code at that address by using Kerninst. Since an advice body is transformed into a C function in a loadable kernel module, the hook code calls that function with the address of the join point as an argument. More properly, the hook

code calls the trampoline function, and the trampoline function will call the function made from an advice body as will hereinafter be described in detail. Here, we treat that the hook code calls the function made from an advice body instead of the trampoline for ease of explanation. Kerninst substitute a jump instruction for the original instructions at that address. The jump instruction jumps to the hook code given by KLASY, which is placed somewhere else, and the original instructions are executed after the hook code. If the size of the replaced instruction is too small to put the jump instruction (5 bytes), Kerninst uses a breakpoint-trap instruction, which is only one byte. If the thread of control reaches that instruction, a trap handler is invoked and it executes the hook code and the instruction replaced with the breakpoint-trap instruction. Since the breakpoint-trap instruction causes software interruption and hence it implies a larger performance penalty than the jump instruction, Kerninst uses the breakpoint-trap instruction only when the jump instruction cannot be substituted for the original instruction. If it is required to weave multiple advice bodies at the same join point, KLASY generates a trampoline function that calls multiple advice bodies in turn. The hook code calls that function instead of advice bodies. This implementation is due to limitations of the current version of Kerninst. Kerninst cannot insert more than one hook code into the same execution point for executing several hook code in turn.

4.3.3 Context exposure

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

To obtain the value of a local variable or a function argument, KLASY reads the .debug_info section of the kernel binary to know the register number or the memory address of that variable. The .debug_info section is generated by the -g option to the gcc compiler. The -fno-omit-frame-pointer option is also used to remain a frame pointer to know an accurate position of a local variables. Since the frame pointer management is required and frame pointer cannot be used as regular register, the -fno-omit-frame-pointer option will cause performance degradation. With the -fno-omit-frame-pointer

option, the value of esp register is copied to ebp register at the beginning of each function. We do not think this is a big drawback because the same performance degradation will be caused if the developers use debugger. That is because debugger, for example gdb, cannot get proper value of local variables without the frame pointer. If a register is allocated to a variable, it is saved by Kerninst on stack memory before a trampoline function is called and thus KLASY reads the stack memory to obtain the value of such a register variable. On the other hand, if a stack frame is allocated to a variable, KLASY reads the saved value of ebp register to obtain the address of that stack frame and computes the memory address for a variable based on that.

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

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

If these accesses are selected by a pointcut, the addresses of the target structures are the values of &inode and inode_ptr. Our extended gcc compiler generates extra symbol information of how those values are computed from local variable inode and inode_ptr. We modified the parser of gcc to traverse abstract syntax tree made by the parser when the member access occur. KLASY computes the addresses according to that symbol information.

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

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

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

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

However, our compiler does not generate such symbol information if an

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

Figure 4.4. An example of execution pointcut

access is something like this:

current_thread_info()->exec_domain

since current_thread_info() is a function call. Since calling it twice may cause unexpected side effect, our compiler will avoid this. Our compiler generates symbol information for computing a target address only if the address is a member of a structure directly or indirectly referred to by a local variable. If the target address is computed by an expression that KLASY does not support, our compiler reports an error.

4.3.4 Execution pointcut

KLASY also support the execution pointcut designator, which select function execution as a join point. For example, Figure 4.4 is an aspect that prints a trace message when a function check_free_space is executed. Note that wild card can also be used with the execution pointcut, and the execution pointcut can be combined with other pointcut by using AND and OR as explained before.

The implementation of the execution pointcut is simple because the symbol information generated by a normal C compiler (in our case, gcc) includes the memory address of function entry points. KLASY uses Kerninst for inserting hook code, which calls an advice body when executed. If an advice is a before advice, KLASY inserts the hook code at the entry point of the specified function, that is, at the beginning of the function body. If an advice is an after advice, KLASY inserts the hook code at the exit point of the function. This insertion is processed by Kerninst. Note that both the entry

point and the exit point of the specified function are calculated by Kerninst. Due to the limitations of Kerninst, the execution pointcut cannot be used with a static function or an inlined function. Again, only a pointcut used with the execution pointcut is the argument pointcut, and no other pointcuts can be used with the execution pointcut.

4.3.5 Unweaving

KLASY supports unweaving of an aspect during runtime. KLASY records all the aspects that have been woven and, when the users request KLASY to unweave one of the aspects, KLASY removes the hook code inserted for that aspect, or removes the calls of the advice bodies from the trampoline functions. The modification of the binary code is processed by using Kerninst. KLASY updates all the trampoline functions and advice bodies, compiles them to the loadable kernel module, replaces the loadable kernel module, and inserts hooks by Kerninst again. The users can run a command for unweaving an aspect with the name specified by a command-line argument. For example,

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

unweaves an aspect named inode_trace.

4.4 Summary

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

Our dynamic weaving technique uses a modified C compiler that generates extended symbol information, which the dynamic weaver refers to for finding the memory addresses of the join points (shadow) selected by pointcuts. The dynamic weaver modifies the binary code of a running kernel so that hook code for executing an advice body is embedded at those addresses. The extended symbol information also enables accesses to target structures.

The contributions of this study are to present the source-based binary-level dynamic weaving and to discuss limitations of that approach. One drawback of that approach is that our modified gcc compiler does not optimize as well as the original gcc since our compiler must generate extended symbol information. We will do some experiments for measuring overheads caused by this drawback. Another drawback is that an aspect weaver may fail to find some join points selected by a pointcut if a compiler performs serious code motion for optimization.

Chapter 5

The Xflow Pointcut and XenLASY

In this chapter, we propose XenLASY [79], which is a dynamic AOP system providing the xflow pointcut. The xflow pointcut is a pointcut for selecting a data flow manually defined by the developers. Since OS kernels are multi-threaded applications, support of the pointcut is important. With the pointcut, the developers can keep tracing even after a thread manipulating data changed. The developers can define where to start tracing, where to transit tracing, and where to quit tracing. According to the definition, XenLASY will automatically start, propagate, and end a data flow. Without the xflow pointcut, the situation become difficult. Conceivable alternates are using a cflow (aka. control flow) and a naive data flow. The developers cannot keep tracing with cflow after a thread changes. The developers cannot keep tracing with a naive data flow after a structure of data changes. Note that each flow has its own ID number, and the developers can distinguish one flow from another.

Another benefit of the xflow pointcut is that the developers can avoid logging unnecessary data. With the xflow pointcut, the developers can select data that is related to a data flow they defined. It is done automatically by XenLASY according to the definition. With this feature, amount of memory needed for logging can be reduced.

A remarkable feature of XenLASY is that it supports tracing between

virtual machines (VMs) through the Xen virtual machine monitor (VMM). With this feature, the developers can keep tracing from Domain U to network via Domain 0. Note that Domain 0 is a privileged VM that can handle physical devices, while Domain U is a VM that request I/O for Domain 0. For implementing this feature, the ID number attached to each flow is propagated with data through the Xen VMM. It is also automatically done by XenLASY. The way of propagation is also defined by the developers.

The rest of this chapter is organized as follows. Section 5.1 explains why the xflow pointcut is required in details. Section 5.2 shows syntax and usage of the xflow pointcut. Section 5.3 shows how to define the xflow pointcut. Section 5.4 explains an implementation of the xflow pointcut. We compare our study with other systems in Section 5.5, and we conclude this chapter in Section 5.6.

5.1 Necessity of the xflow pointcut

Performance tuning used to work well inside an OS kernel because all the devices and user processes are administrated by it. All bottlenecks used to be fixed by modifying an OS kernel. There was a performance improvement technique for a disk I/O named anticipatory scheduling [37]. This technique will delay requesting a read operation to reduce disk-seek overhead.

After virtual machine (VM) becomes widely used, performance tuning should be done not inside OS kernels but whole system including each VM and VM monitor (VMM). That is because hardware is shared by several VMs. In this case, even if the developers tune up one of the VMs, it will not cause performance improvement of whole system. Moreover it will not improve performance of the VM. For example, anticipatory scheduling, stated above, will not work well on VM [38]. Although anticipatory scheduling need accurate estimation of a position of a disk head, it is difficult to do for an OS kernel on VM. This will cause frequent seeking of the disk head.

Performance tuning that considers Domain 0 is required for the Xen virtual machine monitor [8] in particular. Domain 0 is a privileged VM that can manipulate a real hardware. All the other domains, called Domain U, will request I/O operation to Domain 0 as showed in Figure 5.1. Since the entire request is passed from Domain U to Domain 0, the number of bottleneck candidates will increase. For example, a process of requesting I/O operation from Domain U to Domain 0, scheduler of VMs, a process of requesting I/O operation to a real hardware in Domain 0 can be bottlenecks. In fact, there is

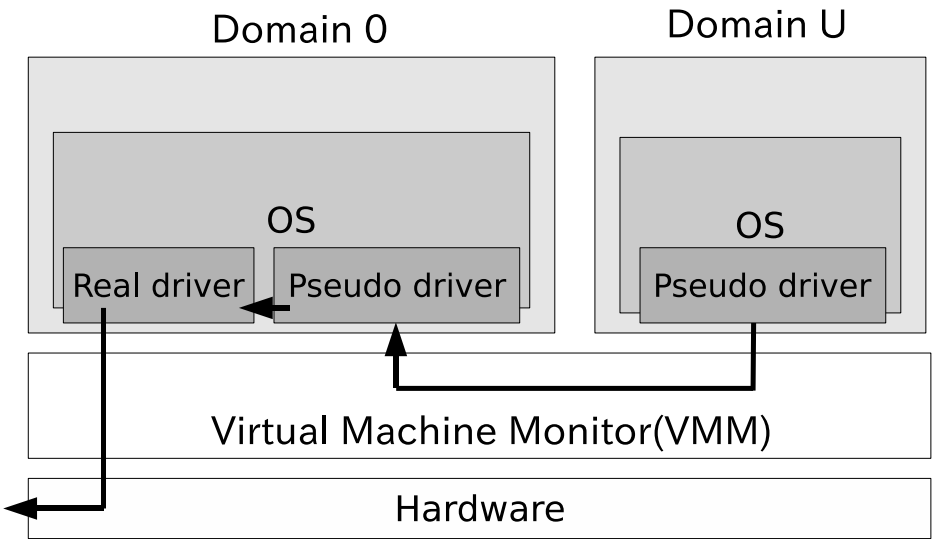

Figure 5.1. I/O flow of Xen

a report that shows the process of requesting I/O operation from Domain U to Domain 0 is a bottleneck [53]. If more than one Domain U publish I/O request, congestion will occur among these domains. Then, each domain should wait for their turn.

To investigate performance bottlenecks under this situation, performance profiler should be able to trace I/O flow from Domain U to Domain 0. Since bottleneck candidates can exist among whole system, the developers should want to investigate performance bottlenecks of each flow. If they can distinguish a flow from the other flows, they can save their trouble. However, call flow (aka. cflow or control flow) is not sufficient for this use. I/O flow among some domains cannot be traced by call flow. Moreover, since I/O manipulation inside an OS kernel is done at two parts, top-half and bottom-half, I/O flow cannot also be traced this point. Note that top-half is an I/O manipulation to provide I/O system call for a user land process and bottom-half is an I/O manipulation to manage a real device. They are executed in different threads to minimize a time for interrupt handler execution.

On the other hand, a naive approach of tracing data flow is not sufficient for tracing I/O flow from Domain U to Domain 0. The approach cannot chase I/O flow if a data structure of the I/O flow changes. A data structure will change from sk_buff, which is used inside the Linux kernel, to raw data in shared memory, this approach cannot trace I/O flow between domains. The

naive approach cannot trace if the data of the flow is copied or separated. For example, if copy of the data is made for retransmission of TCP or the data is separated to smaller size for realize fragmentation, the developers cannot trace the I/O flow.

5.2 The xflow pointcut

To enable profiling over virtual machines, we propose XenLASY. XenLASY provides the xflow pointcut for tracing a data flow of I/O processing on Xen. It will select points where specified data is in a data flow defined by the definition of the xflow The definition of the xflow pointcut specifies where a data flow starts, how a data structure of a data flow is changed, and where a data flow ends. Each data flow is distinguished by the ID number. We call the ID number as the flow ID. XenLASY will automatically assign a unique number to each data flow defined by the definition of the xflow pointcut.

OS kernels are multi-threaded application, and data is manipulated by several threads for achieving purpose inside it. For example, network I/O will be processed at system call parts and device driver parts. They are executed in different threads to make device driver work well. Profiling by tracing a call flow is not sufficient since functions are not called one after another in the kernel for sending or receiving data. Moreover, a call flow will end when the executing thread changes.

Tracing mechanism that can pursuit over domains is needed. Since data is propagated among domains, call flow will be broken at the point where a data is passed to the Xen virtual machine monitor (VMM). Since each I/O request is passed through Domain 0, the developers cannot trace whole flow from a system call to device driver. This is crucial problem to investigate performance bottlenecks. That is because many performance bottleneck candidates exist outside the domain such as Domain 0 and the Xen VMM.

Providing a proper abstraction is important for AOP development. AspectJ provides the cflow pointcut that is similar to the xflow pointcut. The cflow pointcut selects a span that starts from a calling of a function and finishes at the returning of the function. On the other hand, the xflow pointcut selects a data flow that manipulates specified data. Since the xflow pointcut if for tracing a data flow, developers can keep on tracing even after a thread that manipulate the data changes. Moreover, if developers describe the way clearly, the xflow pointcut enables them keep tracing even if a VM changes. It is possible for developers to do the same thing without the xflow pointcut

by manually writing a code by hand. (Note that it is difficult to write simple macros for this. We will write about this after) However, the developers can simplify the profiling code by using the xflow pointcut if they define the definition of the xflow pointcut once.

Figure 5.2 is an example of aspect using the xflow pointcut. Pointcuts are written inside a block of pointcut elements and codes are written inside a block of before and after elements. In this case, a developer uses the flow named netflow, which is a data flow of network I/O. The access pointcut in Figure 5.2 select an execution point where any member (%) of sk_buff structure is accessed. An advice code will be executed if the selected structure instance is data of a data flow whose name is netflow.

As shown in above, selection of a data flow can be done by passing a name of a flow and a name of a variable as arguments of the xflow pointcut. Its syntax is:

xflow(*flow name, variable name, flow id*)

In this example, reference to the sk_buff structure instance is got by the target pointcut, and it is named as skb. If skb is a data of the netflow data flow, advice code will be executed. Each flow can be identified by a unique ID number. The ID number can be obtained as a variable whose name is specified by 3rd parameter of the xflow pointcut. Note that this parameter can be omitted if developers do not use the ID number. At this time, syntax will be:

xflow(*flow name, variable name*)

For example, developers can write:

xflow(netflow, skb)

to select a data of the netflow flow.

Advice code in Figure 5.2 will store a program counter (pc) of the selected point, a flow id (id) and a time stamp (tsc). Note that DO_RDTSC is a macro to get a value of current time stamp counter, and STORE_DATA3 is a macro to store given data to kernel memory for reading from userland process later. There is a series of STORE_DATA* family to store some variables to kernel memory; for example, STORE_DATA1 for storing one variable, and STORE_DATA2 for storing two variables. Since STORE_DATA* family will

```
<advice>
  <pointcut>
    access(sk_buff.%) AND target(skb) AND
    xflow(netflow, skb, id)
  </pointcut>
  <before>
    long long tsc;
    DO_RDTSC(tsc);
    STORE_DATA3($pc$, tsc, id);
  </before>
</advice>
```

Figure 5.2. An aspect using xflow pointcut

first get a spin lock before storing data, conflict of each STORE_DATA* for storing data will be avoided. By using this, developers can know when and where data of a specified flow passed later.

5.3 The definition of the xflow pointcut

Figure 5.9 is an example of the definition of the xflow pointcut. Note that we also call this definition as the xflow pointcut definition. The flow of the example traces from alloc_skb_from_cache function, where structure instance is made, via skb_clone function, where it is copied, to __kfree_skb function, where structure instance freed. Developers can define the definition of the xflow pointcut with a description like the example. It will be written separately from the advice block. The name of each xflow is given by name attribute, and will be used for selecting the definition of the xflow pointcut in advice. Developers can write any keywords here. For example, developers want to name it as disk_io, the description will be:

```
<xflow name="disk_io">
```

There are rules that are written in the definition of the xflow pointcut to specify a start point (the start element) and an end point (the quit element). The transit element is also used to specify a relay point for keeping tracing after a data structure changes. By using these rules, tracing can be continued after changes of threads, domains, and data structures. Moreover, logging unnecessary data will be avoided by precisely writing the rules. At the start

```
<start>
  <pointcut>
    access(sk_buff.data) AND
    within_function(netif_receive_skb)
  </pointcut>
</start>
```

Figure 5.3. An example of the start element

and end points, the developers specify data and position for tracing by writing pointcuts. At the beginning of tracing, a pair of a pointer to structure instance and newly created ID number is stored to a table. Note that structure instance is specified by a pointcut. At the end point, the pair is removed from the table not to be able to get the ID number from the pointer to the structure instance. If developers want to start a data flow from accessing data member of sk_buff structure in netif_receive_skb function, they will write the start element like shown in Figure 5.3. Although this code do not have a code snippet for registering the flow ID number to the table at a glance, XenLASY automatically add the code for registration.

The developers can write the transit element in the definition of the xflow pointcut to enable developers to get the flow ID number even after a copy of data and a change of data structure. For example, since sk_buff instance will be copied for retransmission of a TCP packet, the copy cannot be traced by simple tracing of a data flow. The transit element will solve this problem. By writing this, an ID number of a structure instance of copy source will be stored to the table as the ID number of a structure instance of copy destination. Both copy source and destination are specified by the names of local variables that contain structure instances in them. We will write details later.

Selection of these points can be done with a pointcut as we showed before. An instruction for specifying a structure instance to store into or remove from the table can be omitted in the start element and the quit element. If it is omitted, the structure instance specified by the access pointcut will be selected. To select the other variables, developers should write the select element to specify the variable that contains a structure instance. For example, selecting the local variable named data will be written as:

```
<select local_var="data" />
```

If developers want to use data variable in alloc_skb_from_cache function, the

```
<start>
  <pointcut>
    access(sk_buff.data) AND
    within_function(alloc_skb_from_cache) AND
  </pointcut>
  <select local_var="data" />
</start>
```

Figure 5.4. An example of the select element

```
<start>
  <pointcut>
    access(sk_buff.data) AND
    within_function(netif_receive_skb) AND
    target(skb)
  </pointcut>
  <action>
    ... some code for registering ID number ...
  </action>
</start>
```

Figure 5.5. An example of the action element

code will be in Figure 5.4.

To write a registration code by themselves instead of using normal way of registration, there is the action element to specify the code. Figure 5.5 is the example for using the action element. If developers use the action element, they should add a pointcut designator to get a local context. target(skb) is used to get a local context in Figure 5.5. The code developers can write inside the action element is the same as that of the before or after advice. Using the action element is not admirable because all the code should be written by themselves, which is to annoying to do. The action element is also supported by the transit element and the quit element. Figure 5.6 and Figure 5.7 are examples. As the same as the action element used with the start element, the developers should write the target pointcut or the local_var pointcut to get local contexts by themselves.

The move element in the transit element specifies a source and destination variable name of structure instances for propagating a flow ID number. In this example, since skb variable is written in the from attribute and n variable is written in the to attribute, the flow ID number associated with the skb variable will be succeeded by n variable at the specified pointcut. After propagation, an entry that represents a pair of the passed ID number and

```
<transit>
  <pointcut>
    access(sk_buff.data) AND
    within_function(netif_receive_skb) AND
    target(skb) AND local_var(data)
  </pointcut>
  <action>
    ... some code for registering ID number ...
  </action>
</transit>
```

Figure 5.6. An example of the action element with the transit element

```
<quit>
  <pointcut>
    access(sk_buff.data) AND
    within_function(netif_receive_skb) AND
    target(skb)
  </pointcut>
  <action>
    ... some code for registering ID number ...
  </action>
</quit>
```

Figure 5.7. An example of the action element with the quit element

skb variable will be removed from the table. If developers want to remain the entry, they should use the copy element instead of the move element. If they use the copy element, the ID number will be got from both skb variable and n variable after propagation. Figure 5.8 is an example using the copy attribute. In the example, since the copy element is used instead of the move element, the ID number can be resolved from structure instances in both skb variable and n variable after execution of skb_clone function. Note that the transit element should have either the move element or the copy element in it but cannot have both. If syntax is wrong, an error will occur.

5.3.1 The definition for inter-domain transit

We provide the xin_move, xin_copy, xout_move and xout_copy elements to enable propagation of an ID number over the Xen VMM. They can be written inside the transit element. We showed usage of the transit element in Section 5.3, which cannot be used for propagating an ID number over the Xen

```
<transit>
  <pointcut>
    access(sk_buff.head) AND
    within_function(skb_clone)
  </pointcut>
  <copy from="skb" to="n" />
</transit>
```

Figure 5.8. An example of the copy attribute of the transit element

VMM. Since an ID number is gotten from a pointer to structure instance, it will not be gotten in a different domain that do not share an address space and the table. Moreover, since a data structure will change dramatically for sending data from a domain to the other, a simple binding mechanism, which will get an ID number from a pointer to a structure instance, will not work well.

Figure 5.11 is an example of the definition of the xflow pointcut for inter-domain transit. This description represents tracing of a network I/O data passed from Domain U to Domain 0. Data used in each domain will be separated to a header part and a data part by a pseudo-device driver when it is passed through the Xen VMM. The data part will be passed by using a shared memory. A header part will represent the position of the shared memory that has the data part. The header part will be passed separately from the data part. In XenLASY, the ID number will be stored inside the header part for propagating it over the Xen VMM. It will be extracted at a destination domain. In this example, the flow ID number got from skb variable will be stored into tx variable at the point where flags member of netif_tx_request structure is accessed in the netfront.c file of the linuxU domain. Note that tx variable is a variable of a header part for network I/O, and linuxU is one of Domain U type domains. The data of tx variable in the linuxU domain is accessed as the data of txp variable in the linux0 domain. Note that the linux0 is the Domain 0 type domain. The ID number stored inside txp variable will become the flow ID number of skb variable at the point where flags member of netif_tx_request structure is accessed in the netback.c file of the linux0 domain. They are instructed by the xin_move and xout_move elements.

The xin_move element represents how a flow ID number is stored to the header part, and the xout_move element represents how a flow ID number is extracted from the header part. The name attribute will be given at the xin_move element. It will be referred by the name attribute in the xout_move

```
<xflow name="netflow">
  <start>
    <pointcut>
      access(sk_buff.data) AND
      within_function(alloc_skb_from_cache)
    </pointcut>
  </start>
  <transit>
    <pointcut>
      access(sk_buff.head) AND
      within_function(skb_clone)
    </pointcut>
    <move from="skb" to="n" />
  </transit>
  <quit>
    <pointcut>
      access(sk_buff.%) AND
      within_function(__kfree_skb)
    </pointcut>
  </quit>
</xflow>
```

Figure 5.9. An example of the definition of the xflow pointcut

element or the xout_copy element later. The name attribute of the xin_move or xin_copy element and the name attribute of the xout_move or xout_copy element should be in concord. For example, if developers write:

```
<xin_move name="xennet" from="skb" to="tx" >
...omitted...
</xin_move>
```

for sending the flow ID number. Then, they should also write:

```
<xout_move name="xennet" from="txp" to="skb" >
```

later to receiving it. Again, the name field should be in concord. The from attribute of the xin_move element represents a name of a variable that has the ID number to be propagated. The to attribute of the xin_move represents a name of a variable that the ID number will be stored into. The field element, which follows the xin_move element, shows a bit field used for storing the ID number. The offset attribute represents an offset of a bit field and the

```
<xin_move name="netin" from="skb" to="tx" />
  <field name="flags" offset="4" size="12" />
  <field name="offset" offset="4" size="12" />
  <field name="size" offset="4" size="12" />
</xen_move>
```

Figure 5.10. An example of the multiple field elements

size attribute represent a size of a bit field. Following code from Figure 5.11 represents using of flags member from least 4 bit to 16 bit:

```
<field name="flags" offset="4" size="12" />
```

Note that flags member is a member of a structure whose instance is given by the to attribute of the xin_move or xin_copy element. Multiple fields can be used to send the flow ID number if developers write multiple field elements. The flow ID number is stored from lower bit to upper bit. The example for using the multiple field elements is shown in Figure 5.10. In Figure 5.10, the flags member, the size member and the offset member are used to store the flow ID number. They are in netif_tx_request structure instance whose pointer is in tx variable. If the flow ID number does not go in the specified member, a series of upper bits will be ignored. This limitation will not be a big problem since data of the specified data can be distinguished by sorting with a time stamp even if some upper bits are eliminated. The flow ID number assigned to skb variable will be removed at the propagation time if developers use the xin_move element. If they use the xin_copy element, the assignment will be remained. The xout_copy element will also remain the assignment besides the xout_move element will remove it. These mechanisms enable developers to keeping tracing even if domains manipulating the data are changed.

@ is used inside within_file pointcut to specify a domain name where an aspect will be woven. An example is shown in Figure 5.11. As we mentioned before, within_file pointcut will be used to limit a range of join points inside a specified file. With @, it can also limit a range of join points inside the specified file in the specified domain. For example, netfront.c@linuxU means that join points are limited in netfront.c file of linuxU domain. @ is also used with within_function pointcut, which will limit join points to the specified function in the specified domain. If no @ is specified, join points in all domains can be selected. It means that an aspect will be woven into the entire domain.

```
<transit>
 <pointcut>
  access(netif_tx_request.flags) AND
  within_file(drivers/.../netfront.c@linuxU)
 </pointcut>
 <xin_move name="netin" from="skb" to="tx" />
  <field name="flags" offset="4" size="12" />
 </xen_move>
</transit>
<transit>
 <pointcut>
  access(netif_tx_request.flags) AND
  within_file(drivers/.../netback.c@linux0)
 </pointcut>
 <xout_move name="netin" from="txp" to="skb" />
</transit>
```

Figure 5.11. Xflow definition for inter-domain transit

5.4 Implementation of the xflow pointcut

We implemented XenLASY by extending KLASY, which we implemented before. The main points of extension are extending aspect language for supporting xflow, implementation of distribution mechanism, and revision of Kerninst, which we uses to insert a code into a running kernel, for supporting domains on Xen. This section describes these points one by one.

5.4.1 Extension of aspect language

This section shows how the real code made from the xflow definition shown in Section 5.2. Then, we introduce how the xflow pointcut will be translated into normal aspects. Note that code made from the definition of the xflow pointcut will be woven before other normal aspects. Because of this mechanism, the developers do not have to care for the order of aspects weaving even when they use the xflow pointcut.

For example, the start element in Figure 5.9 will become code in Figure 5.12. This aspect first get a pointer to a structure selected by the access pointcut by using the target pointcut, then it is stored into the table with a newly created ID number. For this purpose, the target pointcut will be automatically added to the original pointcuts. Note that get_new_flowid function will generate a newly created ID number and register_flowid function will

```
<advice>
 <pointcut>
  access(sk_buff.data) AND
  within_function(alloc_skb_from_cache)
  AND target(random_string)
 </pointcut>
 <before>
  register_flowid(netflow, random_string,
   get_new_flowid());
 </before>
</advice>
```

Figure 5.12. A code example for the start element

store a pair of a pointer to a structure instance and the flow ID number to let developers look it up from the pointer to the structure instance later. If the select element is used, the local_var pointcut will be used instead of the target pointcut. In this case, a local variable is used as a pointer to a structure.

The transit element in Figure 5.9 will become code in Figure 5.13. This aspect will get the flow ID number assigned to the structure instance in skb variable specified by from attribute, and store a pair of the flow ID number and the structure instance in n variable to the table. References to skb variable and n variable are got by using the local_var pointcut, and *skb and *n are their value. For example in Figure 5.9, following pointcuts are automatically added to the original pointcuts:

```
AND local_var(skb, random_str1)
AND local_var(n, random_str2)
```

Note that get_flowid function is used to get a flow ID number from a pointer to a structure instance. The ID number assigned to the structure instance in skb variable will be eliminated since the move element is used. If developers want to remain the flow ID number, then they should use the copy element instead of the move element. If they use the copy element, remove_flowid(netflow, skb) will not be executed. An example code using the copy element is shown in Figure 5.14.

The first transit element in Figure 5.11 will be translated into code in Figure 5.15. Even for the transit element that has the xin_move element in it, references to the local variables specified by the from attribute and the to

```
<advice>
 <pointcut>
  access(sk_buff.head) AND
  within_function(skb_clone)
  AND local_var(skb, random_str1)
  AND local_var(n, random_str2)
 </pointcut>
 <before>
  void *skb = *(void**)random_str1;
  void *n = *(void**)random_str2;
  int id = get_flowid(netflow, skb);
  if (id != 0) {
    register_flowid(netflow, n, id);
    remove_flowid(netflow, skb);
  }
 </before>
</advice>
```

Figure 5.13. A code example for the transit element

attribute will be got by following pointcut:

```
AND local_var(skb, random_str1)
AND local_var(tx, random_str2)
```

as the same as the transit element is used with the move element or the copy element. To store the flow ID number into a header transferred among domains, the flow ID number is got from the structure instance in skb variable by using get_flowid function, and stored into members of the structure specified by to attributes of the xin_move or xin_copy element. The members used for storing the flow ID number are specified by the field elements in the xin_move element or the xin_copy element. The field elements can occur in multiple times. Note that the flow ID number assigned to skb variable is removed from the table by remove_flowid function because the xin_move element is used. The remove_flowid function will remove an entry of the flow ID number and the pointer to the structure instance from the table. The xout_move element in the 2nd transit element of Figure 5.11 will do a reversed procedure done by the code for the xin_move element. The code for this will get the ID number from a header, then store it to the table at the domain where the flow ID number is read. The code for the transit element with the xout_move element is shown in Figure 5.16. The quit element will be translated to the code similar to the start element. The code will remove the

```
<advice>
 <pointcut>
  access(sk_buff.head) AND
  within_function(skb_clone)
  AND local_var(skb, random_str1)
  AND local_var(n, random_str2)
 </pointcut>
 <before>
  void *skb = *(void**)random_str1;
  void *n = *(void**)random_str2;
  int id = get_flowid(netflow, skb);
  if (id != 0) {
    register_flowid(netflow, n, id);
  }
 </before>
</advice>
```

Figure 5.14. A code example for the transit element used with the copy element

ID number from the table by calling remove_flowid function instead of register_flowid function. The code translated from the quit element in Figure 5.9 is shown in Figure 5.17.

The aspect shown in Figure 5.2 will be translated into the code in Figure 5.18. The aspect has the xflow pointcut in its pointcut element. At the time code using the xflow pointcut is woven, it is translated into the code using get_flowid function. If the flow ID number cannot be got from get_flowid function, advice will not be executed. Note that a random string will be used as the name of the variable that contains the flow ID number if developers omit the third parameter of the xflow pointcut. The random string is long enough that probability of collision between it and other strings is negligible. For example, the aspect in Figure 5.19 will be translated into Figure 5.20.

It is difficult to implement transformation from the xflow pointcut and the definition of the xflow pointcut to an aspect code by simple macro transformation. That is because a flow name used in the xflow pointcut is judged to be effective or not at compile time. The name should be one of the names given by the name fields of the definition of the xflow pointcut. A similar mechanism is used to refer the way to store the flow ID number in xin_move and xin_copy from xout_move and xout_copy. Moreover, it is difficult to implement the transformation by macro transformation since the name of the variable specified the xflow pointcut is also validated at compile time. For validation, pointcuts should be parsed before translation. It also makes

```
<aspect>
 <pointcut>
  access(netif_tx_request.flags) AND
  within_file(drivers/.../netfront.c@linuxU)
  AND local_var(skb, random_str1)
  AND local_var(tx, random_str2)
 </pointcut>
 <before>
  void *skb = *(void**)random_str1;
  struct netif_tx_request *tx = *(void**)random_str2;
  int id = get_flowid(netflow, skb);

  if (id != 0) {
    tx-&gt;flags |= id &lt;&lt; 4;
    id &gt;&gt;= 12;
    remove_flowid(netflow, skb);
  }
 </before>
</aspect>
```

Figure 5.15. A code example for the xin_move element of the inter-domain transit

macro transformation difficult the developers can omit some of code in the definition of the xflow pointcut. That is because XenLASY should remember some status to support this feature. One of the examples for this is that the third parameter of xflow pointcut can be omitted, which should prepare a unique variable name for storing an ID number.

5.4.2 Distribution of aspects

Control of weaving and unweaving aspects can be centralized since aspects are automatically distributed to target domains. Distribution will be done to domains that are specified by pointcuts. Each domain has a runtime program for coping with distribution of aspects and dynamic weaving as shown in Figure 5.21. For weaving, compiled advices and a list of execution points to insert hooks generated from pointcuts will be sent to the runtime on each domain. Then, the runtime will load the compiled advice to the kernel, and the hooks are inserted according to the list. If @ is used in the within_file pointcut or the within_function pointcut, an advice related to this pointcut will be woven only into the domain specified by the @ designator.

```
<aspect>
 <pointcut>
  access(netif_tx_request.flags) AND
  within_file(drivers/.../netback.c@linux0)
  AND local_var(skb, random_str1)
  AND local_var(txp, random_str2)
 </pointcut>
 <before>
  struct netif_tx_request *txp = *(void**)random_str1;
  void *skb = *(void**)random_str2;
  int id = 0;

  id |= (tx-&gt;flags &gt;&gt; 4) &amp; 0xfff;
  tx-&gt;flags &amp;= 0xfff &lt;&lt; 4;
  if (id != 0) {
    register_flowid(netflow, skb, id);
  }
 </before>
</aspect>
```

Figure 5.16. A code example for the xout_move element of the inter-domain transit

5.4.3 Extension of Kerninst

Since Kerninst did not work correctly on an OS in a domain of Xen, we extended it. The bare bones of extensions are following two points. One is about manipulation of the interrupt table. The other is a judgment of a kernel mode.

Kerninst is a dynamic code instrumentation tool. It can modify a code of a running kernel on an x86 processor. It will use a jump instruction (jmp) or a break point trap instruction (int3) as a hook for modifying kernel behavior. If a thread reached the hook, it will execute the code given by Kerninst, and then return to the original code. The jump instruction is used if there is enough space (5 bytes); otherwise the break point trap instruction is used. If the break point trap instruction is used, a program counter will be changed to the beginning of the code given by Kerninst in a trap handler, and return to the modified program counter.

Kerninst will modify do_int3 function of Linux, which is a trap handler for int3 instruction for implementing this feature. Kerninst will directly refer the interrupt table at modification time to know the memory address of do_int3 function. However, this operation needs a privilege and each domain cannot execute the operation. The reason why Kerninst reads the interrupt table is

```
<advice>
 <pointcut>
  access(sk_buff.%) AND
  within_function(__kfree_skb)
  AND target(random_string)
 </pointcut>
 <before>
  remove_flowid(netflow, random_string);
 </before>
</advice>
```

Figure 5.17. A code example for the quit element

```
<advice>
 <pointcut>
  access(sk_buff.%) AND target(skb)
 </pointcut>
 <before>
 int id = get_flowid(netflow, skb);
 if (id != 0) {
  long long tsc;
  DO_RDTSC(tsc);
  STORE_DATA3($pc$, tsc, id);
 }
 </before>
</advice>
```

Figure 5.18. A code example for the xflow pointcut

that do_int3 function is a static function, whose symbol is not exported. Since Xen do not provide an API to refer the interrupt table, we modified Linux kernel source code to export do_int3 function. We also modified Kerninst to use the exported symbol information of the function instead of analyzing the interrupt table.

Kerninst executes its code only if the break point trap occurs inside an OS kernel. Since Kerninst judges a kernel execution mode before executing its code, we modified the check routine of Kerninst to support execution inside a kernel in a domain of Xen. do_int3 function is always called at the time the break point trap occurs. Thus, it is also called even if the trap occurs from a user-land program. Kerninst distinguishes whether the trap occurred from a kernel or a user-land by using the CR3 register, which shows current a privilege-level. In native Linux, ring 0 shows a kernel mode and ring 3 shows a user-land mode. However, if running Linux on Xen, ring 0 is used by VMM

```
<advice>
 <pointcut>
  access(sk_buff.%) AND target(skb)
  AND xflow(netflow, skb)
 </pointcut>
 <before>
    ... here is the advice ...
 </before>
</advice>
```

Figure 5.19. The xflow pointcut without the 3rd parameter

```
<advice>
 <pointcut>
  access(sk_buff.%) AND target(skb)
 </pointcut>
 <before>
 int random_str = get_flowid(netflow, skb);
 if (random_str != 0) {
    ... here is the advice ...
 }
 </before>
</advice>
```

Figure 5.20. A code example for the xflow pointcut without the 3rd parameter

and ring 1 is used for a kernel. We modified Kerninst to distinguish ring 1 as a kernel-mode.

5.5 Related Work

There are sophisticated tools that have an ability to trace an I/O flow. A good example is a profiler. There are profilers for tracing a data flow over a network, and profilers that supports tracing the Xen virtual machine monitor. Another good example is an AOP system. There are AOP systems for tracing a data flow and that for distributed environment. From next section, we will take a look at some of those tools one by one.

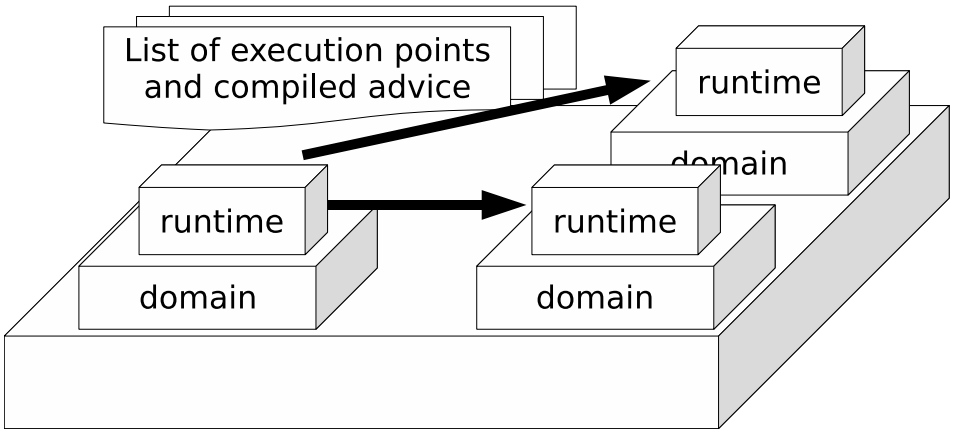

Figure 5.21. How to distribute aspects

5.5.1 Existing tracers

MagPie [9, 7] and CauseWay [14, 13] are good examples of profilers for tracing over multiple modules and hosts. MagPie will store log data at run time for investigating the log data to find performance bottlenecks later. CauseWay is a system to trace a series of I/O flow by passing meta-data with each data. Since XenLASY can store an ID number as meta-data, it is similar to CauseWay. While Causeway needs some extension to a kernel for passing meta-data at run time, XenLASY do not need such source code modification and negligible overhead at run time. While developers using MagPie cannot change behavior by using profiling result at run time, developers using XenLASY can change its behavior at run time.

Pinpoint [15, 16] is a tracer for a distributed environment. It automatically traces a request flow without any definitions. It tags each call with a request ID. This approach is similar to the xflow pointcut. However, its target application is much different. Pinpoint is made for fault analysis by using statistical approach. Besides, the developers using the xflow pointcut store timestamps to investigate performance bottlenecks. Moreover, the target application of Pinpoint is J2EE, and it does not support profiling of operating systems on a virtual machine monitor.

5.5.2 Profilers for Xen

Xenmon [32] and Xenoprofile [53] are well-known profilers for the Xen VMM. There is a case study to increase performance of Xen by using Xenoprofile [52]. Each tool can investigate a performance bottleneck by counting the number of the specified event. An execution point that has a large number of an event is a point of a performance bottleneck. On the other hand, profiling is done based on source code with XenLASY. Since what usage can be thought is that developers roughly find the bottleneck by these tools and go into a detail by XenLASY, these tools and XenLASY are complimentary tools in one another.

5.5.3 Aspect-oriented language

The dflow pointcut [46] is proposed as a pointcut to select a flow of data. The dflow pointcut can select a data flow the same as the cflow pointcut selects a call flow. The dflow pointcut will also select a data flow that is a result of some calculation of data selected by the dflow pointcut. The xflow pointcut cannot be used for profiling OS kernels. Since dflow pointcut will select all the data automatically, performance overheads caused by the pointcut is quite large. Moreover since code snippets for the dflow pointcut are inserted at compile time, it cannot insert a code into a running kernel, which is supported by XenLASY. The dflow pointcut is made for a Java language application, while the xflow pointcut is made for an OS kernel in a domain of the Xen VMM.

There are aspect-oriented system for distributed environment; e.g. DAC++ [3] and DJcutter [55]. Targets of these systems are userland applications written in C++ or Java. On the other hand, XenLASY is an aspect-oriented system for investigating an OS kernel.

5.6 Summary

In this chapter, we proposed a dynamic aspect-oriented system called XenLASY for tracing an I/O processing on Xen. We provide the xflow pointcut to enable developers to write an aspect for tracing a data flow easily. Since the number of modules related to an I/O processing on VMM is much larger than that on an OS kernel, investigation of performance bottlenecks becomes harder on VMM. However, the xflow pointcut will help developers to investigate performance bottlenecks of each I/O flow. By using the xflow pointcut,

developers can trace a flow even if threads or modules that manipulates the flow change.

Chapter 6

Experiments

6.1 Experiments for KLASY

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

6.1.1 Micro benchmarks

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

	Trampoline		
	No	Register	Stack frame
Jump	16	18	19
Breakpoint trap	200	202	203

Table 6.1. Overheads of null advice (nano sec.)

Name	Details
dhry2reg	Dhrystone 2 benchmark using register variables.
whet	Whetstone benchmark.
execl	Performance of exec system calls.
pipe	Throughput of process pipes.
context	Performance of context switching between processes connected through a pipe.
file1	File copy 256 byte.
file2	File copy 1024 byte.
file3	File copy 4096 byte.
create	Process creation.
shell	Shell scripts.
syscall	Overheads of system calls.

Table 6.2. List of benchmarks in UnixBench

Table 6.1 lists the results. We examined three cases: no trampoline function, a trampoline function obtaining data located in a register, and a trampoline function obtaining data located in a stack frame. If an advice is invoked through the jump instruction, an average overhead is about 16 nano seconds per join point. If it is invoked through the breakpoint-trap instruction, an average overhead is about 200 nano seconds. The overhead due to a trampoline function was negligible compared to the overhead of an advice invocation.

6.1.2 Overheads of the KLASY kernel

To evaluate overheads of KLASY in more realistic situations, we ran benchmark programs from UnixBench [75]. The programs are shown in Table 6.2.

We first measured the execution performance of three Linux kernels with these benchmark programs. One is monolithic, which is a kernel compiled by

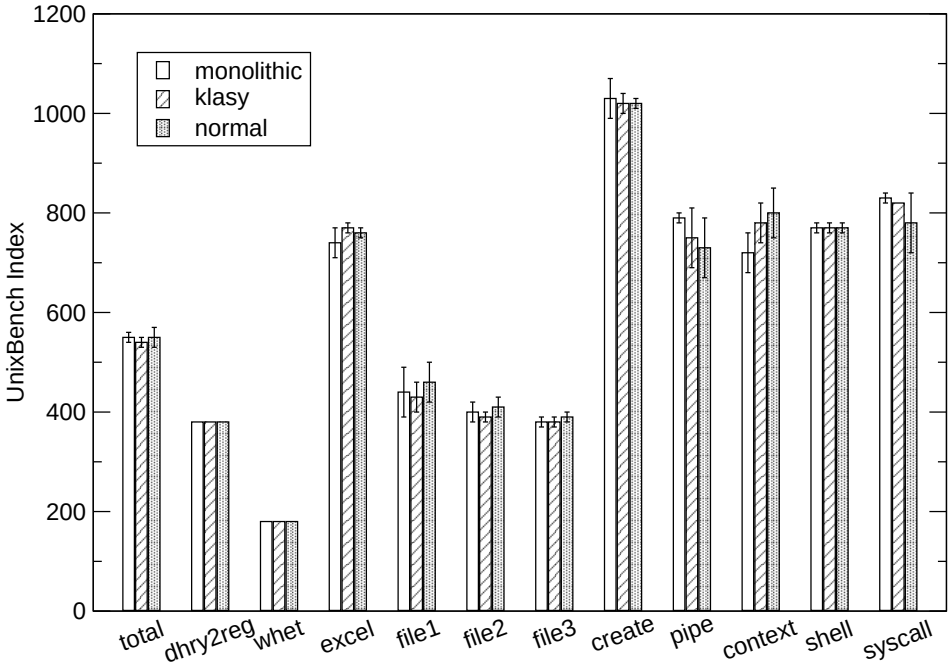

Figure 6.1. The performance indexes of the Linux kernels

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

The results of the benchmarks are shown in Figure 6.1. The results are the index numbers reported by the benchmark programs. A large number is better. According to the results, there is little difference among three kernels. Although the klasy kernel is little slower than the monolithic kernel in several experiments, the average of the overheads is only 1%. The klasy compiler could not optimize as well as the regular compiler because our extended compiler must run with -g and -fno-omit-frame-pointer options as well as -Os option (normal optimization for a kernel) to obtain correct debug information. Those options disable a few optimizations. The other possible cause of the difference is the difference of cache-hit ratio among the tree kernels. That

is because allocation of program code is different among the tree kernels.

6.1.3 Overheads of aspects

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

The results of the benchmarks are shown in Figure 6.2. As the same as the results in Figure 6.1, the results are the index numbers reported by the benchmark programs. A larger number means better performance. The number above each item represents the number of times that the advice body is called at the kernel run time. The number in round brackets ('(' and ')') represents the number of times that the advice body is called by the jump instruction, while the number in angled brackets ('<' and '>') represents the number of times that the advice body is called through the breakpoint trap instruction. We executed the benchmarks for 38 times, and calculated the averages and the variance.

The overheads due to advice execution vary among benchmark programs. They depend on how frequently advice is executed, which instruction (jump or breakpoint-trap) is used for executing advice, and the execution time of the advice body. According to the results, overheads of the aspects are acceptable except the advice body is called for extremely large number of times, such as more than a few ten-millions times. For example, some readers might think that the results of syscall benchmark in (a) have a difference between with and without the aspect. However, there is no statically significant difference. We did a t-test under a 10% level of statically significance, and cannot reject null hypothesis that the tree data of the syscall benchmark are not different. We also did a similar test to the results of the pipe benchmark, and no statically significant difference between with and without the aspect.

On the other hand, there is notable difference between with and without the aspect among the results of the context benchmark in (b). We think that

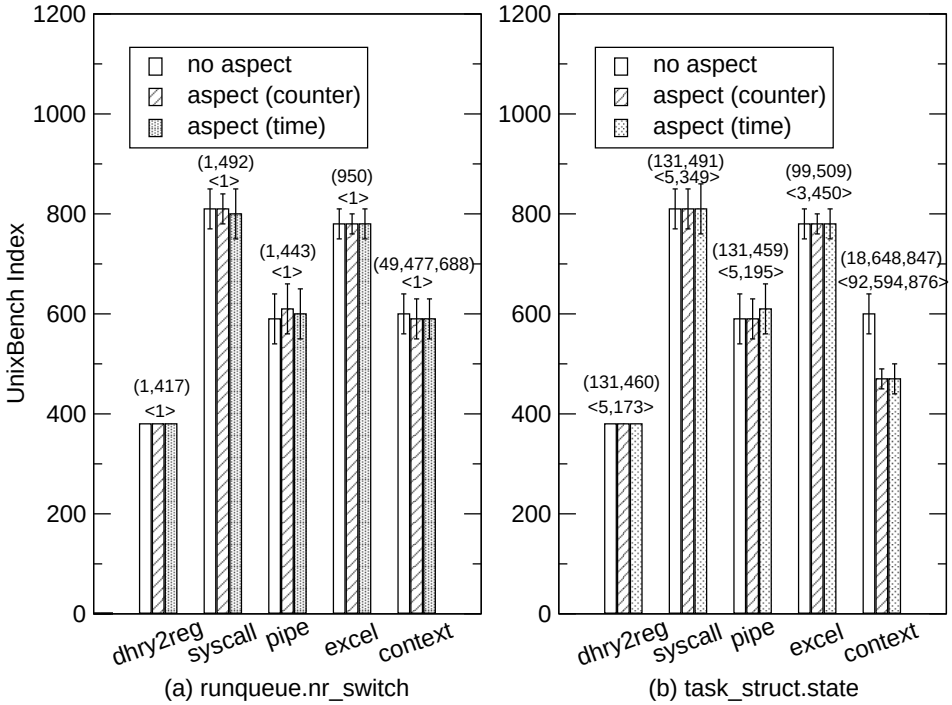

Figure 6.2. Indexes of Unix benchmark

the cause of the difference is that the advice body is mostly called through the breakpoint traps instruction in the context benchmark in (b), while the breakpoint traps are rarely used in other benchmarks. Again, the execution time for calling the advice body through the breakpoint trap instruction is about 200 nano seconds, while the execution time for calling the advice body through the jump instruction is about 20 nano seconds.

6.1.4 Case study

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

Network tracing

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

Figure 6.3 is an aspect we used for this measurement. This aspect pointcuts accesses to all the members of the sk_buff structure and sets arg0 to a pointer to that structure. local.h is a file we wrote. Since some kernel data structures are defined in not a .h file but a .c file, we copied those data structures into local.h and included it in the aspect. The advice first casts the value of target to sk_buff. Then, if the protocol is not ARP, the advice records the current time as well as the program counter pc. DO_RDTSC is a macro provided by KLASY. It executes the rdtsc (Read Time Stamp Counter) machine instruction for obtaining the current time. Its execution time is about 6 nano seconds. STORE_DATA is another macro provided by KLASY. It is used to record data in kernel memory, which can be read later from a user process.

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

We invoked the scp command from a remote host after we wove the aspect. For each arrival of a network packet, we could measure the elapsed time from when the network device of the target host received a packet, at several points

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

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

Figure 6.3. An aspect example for tracing network I/O

of the network sub-system. Table 6.3 shows the result of tracing network I/O. We selected only 11 points out of measured 74 points and two different traces due to the space limitation of the paper. If the target host receives a packet, the thread of control first passes through line 2773 in e1000_main.c. For both traces, it takes 14 to 15 micro seconds from this line to the line 4355 in tcp_input.c. However, the elapsed time from this line to the line 234 in datagram.c is largely different: 12 or 688 micro seconds. When we examined source code, we found that tcp_rcv_established puts sk_buff on a queue and skb_copy_data_iovec dequeues it. Since skb_copy_data_iovec is executed by a process, the time between these two lines depends on process scheduling.

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

Table 6.3. Tracing result of network I/O (partial)

Tracing process switching

In our previous study, we examined how frequently an OS kernel switches processes under heavy workload, and revealed that behavioral anomaly between light- and heavy-weight processes under heavy workload is due to the thread scheduling policy in Linux [35]. To investigate that, we measured a CPU time quantum consumed by each thread in the Tomcat web application server [5], but we had to modify the kernel source code of Linux by hand since the execution point where we wanted to record the time in the scheduler was a member access to a structure. The previous aspect-oriented systems similar to KLASY did not enable us to pointcut member accesses.

If we used KLASY, such measurement could be implemented without modifying the kernel source code. We show the aspect for that measurement in Figure 6.4. This aspect pointcuts accesses the timestamp member of the task_struct structure within function bodies defined in sched.c. The advice body stores the program counter, the process identifier, and the current time. When we wove this aspect, the weaver could successfully insert hook code at 10 lines.

We ran both light- and heavy-weight services on Tomcat after we wove the aspect. To compute CPU time quantum from the log recorded by the advice body, we selected log entries related to process switches (line 2682 in sched.c). Table 6.4 shows the distribution of CPU time quantum consumed by threads. This shows that there are two peaks and the second peak is between 10 and 12. Based on the same observation, we showed that the

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

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

Figure 6.4. An aspect example for tracing process switching

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

Table 6.4. Distribution of CPU time quantum

large CPU time quantum prevents the execution of the light-weight service in our previous study [35]. To perform our previous study, a few more aspects are needed in addition to the aspect in Figure 6.4. Those aspects are not shown here due to limited space but writing them is as easy as writing the aspect in Figure 6.4.

To investigate the overhead of advice execution in a real application, we measured the throughput of Tomcat using the ApacheBench benchmark program [4]. Table 6.5 shows the throughput (the number of processed requests per second) for light- and heavy-weight services with or without the aspect in Figure 6.4. In case of Tomcat, the overhead due to using aspect was negligible even if an advice was executed whenever a process switch occurred.

	Light-weight service	Heavy-weight service
Without aspect	650.20	6.87
With aspect	645.58	6.84
Overhead	0.7%	0.4%

Table 6.5. Throughput of Tomcat (requests/sec)

6.1.5 Effectiveness of the access pointcut

We used the access pointcut in all case studies above. For the C language, selecting join points by specifying a member of a structure is easier than selecting those by specifying functions in most cases even if the developers can select multiple functions. That is because the C language does not have a mechanism to make a group of a series of functions, such as package and class in C++ and Java. We will discuss availability of the access pointcut through showing the problem without the pointcut.

A code of case study in Section 6.1.4 uses a feature that the sk_buff structure is used among functions of a network I/O subsystem in Linux. By selecting all members of the sk_buff structure, we can recognize behavior of the network I/O subsystem. We can also distinguish a packet from another by registering an address of an instance of the sk_buff structure.

Let us think about getting the same information by only using the execution pointcut, which is the pointcut to select a function execution in existing AOP systems. In this experiment, we get logs at 76 execution points. They are in 21 functions because we sometimes get logs at multiple execution points in the same function. 11 functions of 21 functions are static functions, and 6 functions of 11 static functions are inline functions. Since there is a possibility that the same name is used among the different static functions, there is ambiguity for selecting the static functions. Since signatures of the inline functions are eliminated after compiling, we should disable optimization if we use only the execution pointcut. We are also required to have deep knowledge about a network I/O subsystem of Linux: for example, which function is executed when a packet arrives. If we select some functions that are unnecessary for profiling, we will get useless logs. Moreover, a normal kernel is compiled with the optimized option (-Os), some functions are inlined even if they are not qualified by the inline modifier. Thus, optimization for compiling the OS kernel should be disabled to do profiling shown in Section 6.1.4 only with the execution pointcut. However, the result of investigation with an unoptimized

OS kernel is far from realistic result since the OS kernel is normally compiled with optimization option. We believe that the result would not be useful.

Moreover, using the access pointcut enables fine-grained investigation because logs are generated when any members of the sk_buff structure, which can be existing multiple times in a function, are accessed. Besides, the developers using the execution pointcut can get logs only at a beginning and ending of a function. Another merit of KLASY is that we can avoid storing logs for ARP packets, which is unnecessary for investigation, by using the target pointcut. We can know a packet is an ARP packet or not by using information got through the target pointcut.

A case study in Section 6.1.4 measures elapsed time between context switches. In this case study, we used a feature that the timestamp member of the task_struct structure is updated after a process is context-switched. We select the timestamp member access of the task_sturct structure to get a precise timestamp of a context switch. It is roughly possible to do the same thing by using the execution pointcut. However, we cannot select the sched_info_switch function, which really achieve a context switch in the Linux kernel, because it is a static inline function. As mentioned before, a signature of a static inline function is eliminated at compile time. We should select the schedule function instead of the sched_info_switch function. However, since the schedule function is a large code size, accuracy of measuring time quantum of context switches by using the execution pointcut is a little bit lower than that by using the access pointcut.

6.2 Experiments for XenLASY

To investigate XenLASY availability, we did some experiment. Experiment includes micro benchmarks and some case studies. Micro benchmarks are used to evaluate overheads of XenLASY and it works in quite acceptable time. Case studies will show availability of XenLASY in a real situation. A machine used for experiment is AMD Athlon™ 64 3500+ (2.2GHz) and 2GB memory. We used Xen 3.0.4, CentOS 4.4 (Linux 2.6.16.33), gcc 3.3.3, and binutils 2.16 for this experiment. Note that both Domain U and Domain 0 are using the same Linux distribution and the same kernel.

Function name	Details
get_new_flowid	Generate a new ID number.
register_flowid	Register a pair of a pointer and an ID number.
get_flowid	Get an ID number from a pointer.
remove_flowid	Remove an ID number associated to an ID number.

Table 6.6. Functions used inside XenLASY

Function	Elapsed time
get_new_flowid	3 ± 0.0
(empty) get_flowid	9 ± 0.0
(changeID) register_flowid	33 ± 3.0
(changeID) get_flowid	15 ± 1.0
(changeID) remove_flowid	32 ± 2.0
(sameID) register_flowid	33 ± 4.0
(sameID) get_flowid	15 ± 1.0
(sameID) remove_flowid	32 ± 2.0

Table 6.7. Execution time of each function (nano sec.)

6.2.1 Micro benchmarks

We first did micro benchmarks to measure execution time of functions made for implementation of the xflow pointcut and its definition. We measured execution time of each functions in 2000 times. We used the time stamp counter (TSC) to get a precise time during the benchmarks. We did this measurement for 100 times and divided the result by 2000×100 to get the averages. Note that since a kernel module for CPU frequency control was not running during the experiments, CPU frequency is constant during the experiments.

We measured execution time of functions in Table 6.6. We first measured the execution time of get_new_flowid function and get_flowid function with no entry stored (marked as empty). We looked up no existent entry with get_flowid function. Then, we measured register_flowid, get_flowid and remove_flowid in those two situations. One is incrementing an ID number and an address of a pointer (changeID) for each entry, the other is storing different addresses to the same ID number (sameID).

The result of the experiment is shown in Table 6.7. Execution time of each function is acceptable for practical use. Execution times are almost the

same between changeID and sameID. In both case, registration needs only 33 nano seconds and elimination needs only 32 nano seconds on average. 15 nano seconds on average are needed to looking up an ID number from a pointer to a structure instance.

6.2.2 Case study

We investigated a flow of data sent from Domain U to a network via Domain 0 by using xflow pointcut. We also investigated how much resource can be reduced by using the pointcut. The aspect we used is like Figure 5.2 and the definition of the xflow pointcut is like Figure 5.9 and Figure 5.11.

The result of investigation is shown in Table 6.8. The elapsed time item in Table 6.8 shows elapsed time from execution of line 234 in net/core/skbuff.c file. The filename, line number, function name are analyzed from the saved program counter. Program counters are saved by using the aspect when they are executed. According to the result, we can see data is made in tcp_make_synack function, moved to Domain 0, sent to a network in SkGeXmit function, and freed in FreeTxDescriptors function. From the result, the bottleneck of the network I/O is between netif_rx function and netif_receive_skb function in Domain 0. netif_rx function inserts a packet data to a queue for passing it to the top half from the bottom half, and netif_receive_skb will take the packet from the queue. Surprisingly, elapsed time between netif_rx function and netif_receive_skb function is much longer than the elapsed time to pass a packet from Domain U to Domain 0.

Next, we investigated how much resource for investigation can be saved by using the xflow pointcut. At the experiment shown above, the aspects weed out data by using the xflow pointcut. It means that we did not save data that is not registered in the table. We used get_flowid function to distinguish each data is stored or not. To investigate how much resource saved, and how much time saved, we did an experiment with the aspect that does not limit data by using the xflow pointcut. The experiment we did is giving a high overload by using ApacheBench. Note that the saved data is a program counter of the join point, a time stamp, and an address of a pointer to a structure instance. At the experiment, ApacheBench gave 300 requests with 10 requests at a time.

The result of the experiment is shown in Table 6.9. Domain 0 and Domain U shows the memory usage of each domain. The result shows that about 60% memory is saved by using the xflow pointcut. Besides, performances of replying requests are almost the same between with and without

Elapsed time (μs)	Domain	File name	Line number	Function name
0.0	U	skbuff.c	234	alloc_skb_from_cache
3.4	U	tcp.c	726	tcp_sendmsg
15.3	U	tcp_output.c	495	tcp_set_skb_tso_segs
72.4	U	dev.c	1379	dev_queue_xmit
101.3	U	skbuff.c	471	skb_clone
130.8	U	netfront.c	963	network_start_xmit
882.5	0	netback.c	1058	netbk_fill_frags
894.9	0	dev.c	1543	netif_rx
3310.4	0	dev.c	1730	netif_receive_skb
3332.7	0	br_netfilter.c	418	br_nf_pre_routing
3352.2	0	br_forward.c	70	__br_forward
3362.5	0	br_netfilter.c	760	br_nf_post_routing
3367.1	0	br_forward.c	35	br_dev_queue_push_xmit
3368.4	0	dev.c	1379	dev_queue_xmit
3462.6	0	skge.c	1416	SkGeXmit
3495.5	0	skge.c	1829	FreeTxDescriptors

Table 6.8. Result of Tracing

	Domain 0	Domain U
With xflow pointcut	1,557,248	10,587,776
Without xflow pointcut	13,598,976	15,739,968

Table 6.9. Difference of memory usage with or without xflow pointcut (bytes)

the xflow pointcut. The performance is about 382 requests per second in both cases.

6.2.3 Effectiveness of the xflow pointcut

We used the xflow pointcut in case study above. Since data is manipulated by multiple threads in an OS kernel, supporting a pointcut that helps the developers to trace a data flow is important. We will discuss availability of the xflow pointcut through showing the problem without the pointcut.

First, the xflow pointcut helps the developers to write tracing code easily. Since the definition of the xflow pointcut and the xflow pointcut are translated

into aspect codes, the developers can write those aspect codes by hand. However, writing those codes is too annoying to do by hand. As you can see in Chapter 5, codes for managing a data flow contains similar code in too many times. It is error-prone to write them by hand. Moreover, it is difficult to reuse those codes for managing a data flow because those codes are not modularized. On the other hand, the definition of the xflow pointcut is reusable. It has a name to be specified in the xflow pointcut.

The xflow pointcut will also reduce amount of memory required for logging because the developers can avoid logging unnecessary data. According to the result of the experiment in Section 6.2.2, we could reduce 60% of memory compared to the amount of memory without the xflow pointcut, only using the access pointcut.

6.3 Summary

In this chapter, we first measured the overheads of KLASY and XenLASY. Then, we investigated the availability of these systems by using case studies. From micro benchmarks of KLASY, the hook overhead of the breakpoint trap is about ten times larger than that of the jump instruction. We also measured the overhead of the OS kernel compiled by our modified compiler, and an average overhead is 1%.

For KLASY, we measured some overheads of aspects. According to the results of measurement, there is no statically significant difference between with and without aspects in most cases. The exception is that the breakpoint trap is used too much times for invoking the advice body. We also did some case studies; network I/O tracing, investigation of time quantum. Both results show that KLASY is useful for profiling.

For XenLASY, we measured execution time of functions for implementing the xflow pointcut. According to the results, execution times of those functions are acceptable. In all functions, execution time is less than 40 nano seconds. We also did case study for tracing network I/O from Domain U to Domain 0. We can clearly trace a packet and found the bottleneck was in Domain 0. This case study shows that XenLASY is useful for tracing a data flow on Xen.

Chapter 7

Conclusion

This thesis has discussed a dynamic aspect-oriented system for an OS kernel. We proposed that a dynamic AOP is useful for profiling an OS kernel. For profiling an OS kernel efficiently, we proposed new two pointcuts, the access pointcut and the xflow pointcut. The access pointcut selects a member access of a structure, and is useful for collecting join points using a specified data structure. The xflow pointcut selects a data flow, and is useful for tracing a data flow over domains on Xen.

Contributions

The contributions by this thesis are summarized as follows:

- This thesis proposed using a dynamic aspect-oriented programming (DAOP) for profiling an OS kernel. Profiling by DAOP combines both flexibility of execution points and code for profiling, and abstraction of selecting an execution points for logging and writing code.
- This thesis clears that existing DAOP systems for C language lack data-driven pointcuts required for profiling OS kernels. Since they do not use richer symbol information, their pointcuts are limited.

- This thesis proposed the access pointcut, which can specify a member access of a structure as a join point. While existing DAOP for C language cannot select a member access as a join point, KLASY can select a member access because it uses richer symbol information. We also showed that the access pointcut saves time for enumerating functions related to a module for getting logs.

- We proposed a new implementation technique named source-based binary-level dynamic weaving. This is useful for implementing the access pointcut. It collects richer symbol information at compile time, and use the information at weaving time.

- This thesis proposed the xflow pointcut, which can trace a data flow even through changes of threads and virtual machines. Developers can write the way of tracing in details not to get unnecessary logs.

- Case studies showed that those pointcuts are useful for profiling a real OS kernel. We did two case studies for the access and xflow pointcuts. In both case studies, we can know where the performance bottlenecks are.

Future Directions

Possible future directions of this thesis are follows:

Implementation in other architectures Although technique for implementing KLASY is not depend on an x86 architecture and Linux, proving it is not done. It will help development of other OS kernels on other architectures. However, our system depends on Kerninst, which is made only for Solaris and Linux, implementation of hook insertion system for other architectures are needed.

Support for flow division XenLASY did not support flow division because it has a difficulty on assignment of an ID number. It should show relationship to the original as well as it should be distinguished from others.

Weaving inside VMM XenLASY did not support weaving inside VMM because it was not necessary for profiling a data flow we investigated. However, it will be needed if developers want to investigate other things

such as scheduling. Implementation of the feature will extend the execution points for investigation.

Cooperation with profilers using statistics Profiling by DAOP will get exact information base on the source code. It is useful for getting accurate information caused at execution time. However, this approach is weak for disturbance. On the other hand, profilers using statistics is strong for disturbance but cannot get accurate information. The profilers and our tools should be complementary to each other.

Implementation of user interface for line pointcuts If developers can use line pointcuts, it will strongly help users profiling. Although this feature helps developers for profiling, this feature breaks good modularity. That is why we did not implement the feature.

Bibliography

[1] Sufyan Almajali and Tzilla Elrad. Dynamic aspect oriented c++ for upgrading without restarting. In *Proceedings of the 2004 International Conference on Advances in Internet Technologies and Applications, with special emphasis on E-Education, E-Enterprise, E-Manufacturing, E-Mobility*, July 2004.

[2] Sufyan Almajali and Tzilla Elrad. A dynamic aspect oriented c++ using mop with minimal hook weaving approach. In *Proceedings of the 2004 Dynamic Aspects Workshop (DAW04) at Aspect-oriented software development (AOSD 2004)*, March 2004.

[3] Sufyan Almajali and Tzilla Elrad. Coupling availability and efficiency for aspect oriented runtime weaving systems. In *Proceedings of the Second Dynamic Aspects Workshop (DAW05) at Aspect-oriented software development (AOSD 2005)*, March 2005.

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

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

[6] AspectC++ project. The home of aspectc++. `http://www.aspectc.org/`.

[7] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting*

Systems Design & Implementation, pages 259–272, Berkeley, CA, USA, 2004. USENIX Association.

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[9] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 85–90, Berkeley, CA, USA, 2003. USENIX Association.

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

[11] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[12] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *ATEC'04: Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference*, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association.

[13] Anupam Chanda, Khaled Elmeleegy, Alan L. Cox, and Willy Zwaenepoel. Causeway: operating system support for controlling and analyzing the execution of distributed programs. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, Berkeley, CA, USA, 2005. USENIX Association.

[14] Anupam Chanda, Khaled Elmeleegy, Alan L. Cox, and Willy Zwaenepoel. Causeway: Support for controlling and analyzing the execution of multi-tier applications. In Gustavo Alonso, editor, *Middleware*, volume 3790 of *Lecture Notes in Computer Science*, pages 42–59. Springer, 2005.

[15] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based faliure and evolution management. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 309–322, Berkeley, CA, USA, 2004. USENIX Association.

[16] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[17] Y. Coady, M. Feeley, J. S. Ong, and S. Gudmundson. Aspect-oriented system structure. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, page 166, Washington, DC, USA, 2001. IEEE Computer Society.

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

[19] Yvonne Coady, Gregor Kiczales, Michael J. Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *ESEC / SIGSOFT FSE*, pages 88–98, 2001.

[20] Yvonne Coady, Gregor Kiczales, Mike Feeley, Norm Hutchinson, and Joon Suan Ong. Structuring operating system aspects: using aop to improve os structure modularity. *Commun. ACM*, 44(10):79–82, 2001.

[21] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98. ACM Press, 2001.

[22] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[23] Ian Dowse and David Malone. Recent filesystem optimisations on freebsd. In *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*, pages 245–258, Berkeley, CA, USA, 2002. USENIX Association.

[24] Michael Dunlavey. Performance tuning with instruction-level cost derived from call-stack sampling. *SIGPLAN Not.*, 42(8):4–8, 2007.

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

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

[27] Michael Engel and Bernd Freisleben. Toskana: A toolkit for operating system kernel aspects. *Transactions on Aspect-Oriented Software Development II*, 4242:182–226, 2006.

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

[29] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. Slic: an extensibility system for commodity operating systems. In *ATEC'98: Proceedings of the Annual Technical Conference on USENIX Annual Technical Conference, 1998*, Berkeley, CA, USA, 1998. USENIX Association.

[30] Wasif Gilani and Olaf Spinczyk. Dynamic aspect weaver family for family-based adaptable systems. In Robert Hirschfeld, Ryszard Kowalczyk, Andreas Polze, and Mathias Weske, editors, *NODe/GSEM*, volume 69 of *LNI*, pages 94–109. GI, 2005.

[31] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, 1982.

[32] Diwaker Gupta, Rob Gardner, and Ludmila Cherkasova. Xenmon: Qos monitoring and performance profiling tool. Technical Report HPL-2005-187, Hewlett-Packard Development Company, L.P., October 2005.

[33] Keisuke Hatasaki, Tetsuhito Nakamura, and Kazuyoshi Serizawa. Operating system debugging for running enterprise systems. *IPSJ SIG Notes*, 2003(80):33–39, August 2003.

[34] Michael Haupt, Mira Mezini, Christoph Bockisch, Tom Dinkelaker, Michael Eichberg, and Michael Krebs. An execution layer for aspect-oriented programming languages. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 142–152, New York, NY, USA, 2005. ACM Press.

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

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

[37] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o. *SIGOPS Oper. Syst. Rev.*, 35(5):117–130, 2001.

[38] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *USENIX-ATC'06: Proceedings of the Annual Technical Conference on USENIX'06 Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2006. USENIX Association.

[39] kernel.org. Linux changelog-2.6.23. `http://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.23`, October 2007.

[40] Gregor Kiczales and Yvonne Coady. AspectC. `http://www.cs.ubc.ca/labs/spl/projects/aspectc.html`.

[41] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

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

[43] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the ecos kernel. *SIGOPS Oper. Syst. Rev.*, 40(4):191–204, 2006.

[44] Lucent Technologies. From multics to something else. `http://www.bell-labs.com/history/unix/somethingelse.html`.

[45] D. Mahrenholz, O. Spinczyk, and W. Schrder-Preikschat. Program instrumentation for debugging and monitoring with aspectc++. In *ISORC '02: Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 249 – 256, Washington, DC, USA, 2002. IEEE Computer Society.

[46] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In Atsushi Ohori, editor, *APLAS*, volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2003.

[47] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In *FOAL 2002 Proceedings of Foundations of Aspect-Oriented Languages Workshop at Aspect-oriented software development (AOSD 2002)*, pages 17–26, 2002.

[48] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46 – 60. Springer, 2003.

[49] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *Computer Systems*, 2(3):181–197, 1984.

[50] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: a technique for eliminating most synchronous writes in the fast filesystem. In *ATEC'99: Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference*, pages 1–18, Berkeley, CA, USA, 1999. USENIX Association.

[51] Marshall Kirk McKusick and George V. Neville-Neil. Thread scheduling in freebsd 5.2. *Queue*, 2(7):58–64, 2004.

[52] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in xen. In *Proceedings of the USENIX Annual Technical Conference (General Track)*, pages 15–28. USENIX Association, 2006.

[53] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the xen virtual machine environment. In *Proceedings of the First ACM/Usenix Conference on Virtual Execution Environments (VEE'05)*, June 2005.

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

[55] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut: a language construct for distributed aop. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15, New York, NY, USA, 2004. ACM.

[56] Opersys, Inc. Linux trace toolkit. `http://www.opersys.com/ltt/`.

[57] David J. Pearce, Paul H. J. Kelly, Tony Field, and Uli Harder. Gilk: A dynamic instrumentation tool for the linux kernel. In *TOOLS '02: Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, pages 220–226, London, UK, 2002. Springer-Verlag.

[58] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147. ACM Press, 2002.

[59] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *Proceedings of the 2005 Linux Symposium*, pages 49 – 63, July 2005.

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

[61] Dennis Ritchie. The evolution of the unix time-sharing system. In *Proceedings of a Symposium on Language Design and Programming Methodology*, pages 25–36, London, UK, 1980. Springer-Verlag.

[62] Jeff Roberson. Ule: a modern scheduler for freebsd. In *BSDC'03: Proceedings of the BSD Conference 2003 on BSD Conference*, pages 17–28, Berkeley, CA, USA, 2003. USENIX Association.

[63] Yoshiki Sato, Shigeru Chiba, and Michiaki Tatsubori. A selective, just-in-time aspect weaver. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 189–208. Springer-Verlag New York, Inc., 2003.

[64] Wolfgang Schroder-Preikschat, Daniel Lohmann, Fabian Scheler, Wasif Gilani, and Olaf Spinczyk. Static and dynamic weaving in system software with aspectc++. In *HICSS '06: Proceedings of the 39th Annual Hawaii International Conference on System Sciences*, Washington, DC, USA, 2006. IEEE Computer Society.

[65] Marc Segura-Devillechaise, Jean-Marc Menaud, Nicolas Loriant, Remi Douence, Mario Sudholt, Thomas Fritz, and Egon Wuchner. Dynamic adaptation of the squid web cache with arachne. *IEEE Softw.*, 23(1):34–41, 2006.

[66] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus soft updates: asynchronous meta-data protection in file systems. In *ATEC'00: Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*, pages 71–84, Berkeley, CA, USA, 2000. USENIX Association.

[67] Marc Séura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia L. Lawall. Web cache prefetching as an aspect: towards a dynamic-weaving based solution. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119. ACM Press, 2003.

[68] Sourceware.org. Systemtap. `http://sourceware.org/systemtap/index.html`.

[69] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.

[70] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004.

[71] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the third symposium on Operating systems design and implementation*, pages 117–130. USENIX Association, 1999.

[72] Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *Int. J. High Perform. Comput. Appl.*, 13(3):263–276, 1999.

[73] The DWARF Standards Committee. The dwarf debugging standard. `http://dwarfstd.org/`.

[74] The Paradyn project. Kerninst. `http://www.paradyn.org/html/kerninst.html`.

[75] Tux.Org, Inc. Unixbench. `http://www.tux.org/pub/tux/niemi/unixbench/`.

[76] Felix von Leitner. Benchmarking bsd and linux. `http://bulk.fefe.de/scalability/`.

[77] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *ATEC'00: Proceedings of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*, pages 13–26, Berkeley, CA, USA, 2000. USENIX Association.

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

[79] Yoshisato Yanagisawa, Kenichi Kourai, and Shigeru Chiba. Xenlasy: Aspect-oriented profiler for tracing i/o processing on xen. **情報処理学会論文誌. プログラミング**, 49(1):51–62, 2008.

[80] Yoshisato Yanagisawa, Kenichi Kourai, Shigeru Chiba, and Rei Ishikawa. A dynamic aspect-oriented system for os kernels. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 69–78, New York, NY, USA, 2006. ACM Press.

[81] Yoshisato Yanagisawa, Kenichi Kourai, Shigeru Chiba, and Rei Ishikawa. Klasy: System for source-based binary-level dynamic weaving. **情報処理学会論文誌. プログラミング**, 48(10):176–188, 2007.

[82] Charles Zhang and Hans-Arno Jacobsen. TinyC^2: Towards building a dynamic weaving aspect language for c. In *FOAL 2003 Proceedings Foundation of Aspect-Oriented Languages Workshop at Aspect-oriented software development (AOSD 2003)*, March 2003.