2005－OS－99（23）
2005／5／27

アスペクト指向を用いたカーネルプロファイラ

柳澤佳里[†]　光来健一[†]　千葉滋[†]

我々はソースコードレベルの視点でログを取る実行点を指定できるカーネルプロファイラであるKLASについて提案する。このプロファイラは動的アスペクト指向に基づいており、利用者はC言語で書かれた任意のコードをログ取得のためのコードとして実行することができる。このコードは利用者が指定した点で自動的に実行され、ログを収集する。また、KLASは動的アスペクト指向を用いているので、利用者がカーネルに挿入したアスペクトを変更する際に再コンパイルや再起動を行う必要は無い。KLASは実行時にアスペクトを挿入、削除できるシステムであるが、他のシステムと違いそれをソースコードレベルの視点で行うことができる。例えば、利用者はコンパイルによって失われる構造体のメンバへのアクセスをポイントカットとして抽出することができる。そして、利用者は構造体のメンバへのアクセスに相当する機械語の位置を調べることなくこれを行うことができる。我々はこのような仕組みをコンパイラの出力するシンボル情報を拡張することで実現している。我々はKLASをFreeBSD上でGNU Cコンパイラを使って実装し、我々の手法のオーバーヘッドを確認するために実行時間を測定した。

A Kernel Profiler based on Aspect-orientation

Yoshisato YANAGISAWA,[†] Kenichi KOURAI [†]
and Shigeru CHIBA[†]

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

1. はじめに

オペレーティングシステム (OS) の性能向上というのは昔からOS開発者の間での大きな課題であった。MULTICSのあった時代にUnixが生まれた経緯やLinuxがモノリシックカーネルである経緯などもすべて性能のためだと言われている。近年でもこのような性能向上はOS開発者の間での大きな課題の一つである。実際、最近でもOSでプログラムを実行したときの処理性能の向上を目指してFreeBSDやLinuxのスケジューラの新たな実装が開発されている。

このようなOSの性能の向上を行うにはまず性能劣化を引き起こしている箇所を調べることが重要な課題となる。というのも、性能劣化を引き起こしている箇所を見付け、そこを改善すれば性能の向上を見込めるからである。発見した点における性能の向上にはその点に置けるアルゴリズムや抜本的なコードの変更が必要かもしれないが、その点を発見しないことにはその実行時間の改善による性能の向上が行えない。

OS内の性能が劣化している箇所の発見には通常はカーネルプロファイラが使われる。ここで、カーネルプロファイラは関数が呼び出された回数を調べるというような単純な機能だけでは不十分で、利用者が要求したさまざまなデータが得られなくてはならないと

† 東京工業大学大学院 情報理工学研究科 数理・計算科学専攻
Dept. of Mathematical and Computing Sciences. Graduate School of Information Science and Engineering. Tokyo Institute of Technology

我々は考えている。例えば、我々はネットワークI/Oが同時に行われたときにFreeBSDのネットワークモジュールの性能が劣化する原因を調べている。その際、性能が劣化した原因を調べるには、ネットワークモジュール中の関数が何回呼ばれたかという単純な情報だけでなく、我々が指定した点でのメッセージやタイムスタンプの情報も必要だと考えている。

本論文では我々のカーネルプロファイラであるKLASについて論じる。KLASは動的アスペクト指向システムであり、利用者が指定したOSカーネル内の実行点で指定したコードを実行することができる。利用者が指定するコードはタイムスタンプの記録のためのプログラムだけではない。例えば、その実行点での変数のログを出力するというようなC言語で書かれた任意のプログラムを用いることができる。KLASではOSの実行中に利用者はコードを実行する実行点や実行するコードの内容を変更する事ができる。性能が劣化している原因を調査するときは、その調査が進むにつれて注目すべき変数や位置が変わっていくが、その際に開発者は再起動を避けることができる。

KLASは実行時に指定された点にある命令をブレークポイントトラップで置き換える。これにより、コードを実行する点やコードの内容の変更を再起動なしで実現している。KLASの実行時アスペクト指向の実装は新しい技法であり、コンパイル時に消えてしまうシンボルを指定してのプロファイリングが可能である。これを実現するために我々はCコンパイラを改造し、コンパイル時に拡張したシンボル情報を出力するようにしている。我々はKLASをFreeBSDおよびGNU Cコンパイラ上で実装した。

以下、2章ではカーネルプロファイラの要件について詳しく述べる。そして、3章では我々の動的アスペクト指向を用いたプロファイラであるKLASの概要とその実装について述べ、4章ではKLASのプロトタイプ実装とその性能に付いて述べる。5章にて関連研究について触れ、6章でまとめと今後の課題を述べる。

2. カーネルプロファイラの要件

OSカーネル中で性能が劣化している原因となるコードを特定するにはカーネルプロファイラを使うのが一般的である。特に、任意の2点間の実行にかかる時間を測定することができるカーネルプロファイラは有用である。しかしながら、既存のカーネルプロファイラはカーネルの性能劣化の原因を特定するのには不十分であった。例えば、近年のOSのコードはC言語でオブジェクト指向の考え方を用いて書かれているため、プロファイリングのためのコードの実行点は関数ポインタを用いて関数を呼んでいる実行点であることもある。従来のプロファイラは静的に呼び先が定まる関数の実行点しかサポートしておらず、このような実行点はサポートされていない。以下では以上のことを考慮しつつカーネルプロファイラの要件について述べる。

まず、カーネルプロファイラは利用者が指定した2点間の実行にかかる時間を測定できなくてはならない。そして、カーネルプロファイラはもし必要なら再起動せずに利用者から指定された2点の変更を行えなくてはならない。このような時間を測定する点の変更は重要である。というのも、利用者はプロファイリングを行う際にはまず大きな塊で区切って性能を測定し、そしてその中から性能が悪かった部分を取り出してその中を区切って性能を評価するということを段階的に繰り返す必要があるからである。再起動はそれ自体が時間がかかる作業であるので、プロファイリングを行う単位を変更する度に再コンパイル、再起動を行うようでは開発効率が低下してしまう。また、再起動は全てのメモリイメージやネットワークモジュールの状態を初期化するため、利用者が調べたかった問題の発生した状況が消えてしまうことになる。これは問題の再現が難しい場合には致命的である。さらに、変数に特定の値が入っている場合のみに時間を測るような場合を考えると、プロファイリングのためのコードは利用者が指定、変更できなくてはならない。というのも、これを行うには実行時に変数の値を調べなくてはならないからである。もちろん、この機能は利用者が調べたいと思っている変数のログを取得するのにも利用することができる。

次に、利用者がプロファイリングのためのコードを実行するために指定する点は粒度が細かくなくてはならない。例えば、関数が呼び出されている点だけではなく、構造体のメンバへのアクセスが行われる点についても指定できなくてはならない。プロファイリングのためには関数が呼び出される点だけではなく、構造体のメンバー変数の中に格納されている関数ポインタが呼び出されている点についても調査する必要があるからである。事実、近年のOSはC言語の構造体を利用してオブジェクト指向のポリモルフィズムのような機能を実現しており、構造体のメンバー変数に関数ポインタを入れることで実現している。例えば、read システムコールや write システムコールが呼ばれたときにはOSカーネルの内部ではそれに対応する関数ポインタがよばれる。この関数ポインタは open したときに対応づけられたI/Oデバイスを操作するための構

```
<aspect name="log_mbuf_clean">
 <pointcut>
  <member-access name="ext_free"
    struct="m_ext"/>
 </pointcut>
 <before-advice>
 void* resolve_arg(long eip,long ebp,int argn)
 {
   /* ext_free 関数の n 番目の引数を取得 */
 }
 struct timespec ts;
 nanotime(&ts);
 printf("mbuf_clean@%l,%lld,arg: 0x%x,0x%x\n",
  ts.tv_sec, ts.tv_nsec,
  resolve_arg($eip, $ebp, 1),
  resolve_arg($eip, $ebp, 2));
 </before-advice>
</aspect>
```

図 1 KLAS におけるアスペクト記述

造体にある read/write メンバに関数ポインタとして格納されている。また、VFS でも同様の仕組みが使われており、ファイルシステムに対応した mount などを行う関数を割り当てるのに使われている。4.3BSD の子孫である FreeBSD や NetBSD のネットワークモジュールでも同様の仕組みが使われていて、この機能を用いてネットワークデバイスに基づいた処理をメモリバッファ(mbuf) で行っている。

最後に、プロファイリングの処理のためにおきる影響は小さくなくてはならない。もし、プロファイリングのためにかかるオーバーヘッドが大きければ得られたデータは不正確になるおそれがある。他の場所の実行時間を測定しているときの影響を無くすためにもプロファイリングに必要なデータが得られたらプロファイリングのためのコードは消されるべきである。

このような要件を考えると、少なくとも手動でコードを挿入したり消去したりするような単純な OS カーネルのプロファイリング方法では不十分である。というのも、コードを挿入、消去する度に再コンパイル、再起動が必要であり、人手に頼るのでミスをおかしやすく、我々の要件には満たないからである。

3. KLAS: Kernel Level Aspect-oriented System

上記の要件を満たすため、我々は FreeBSD 5.2.1 上で KLAS(Kernel-level Aspect-oriented System) という新しい動的アスペクト指向システムを開発した。

3.1 KLAS の概要

KLAS は FreeBSD の OS カーネル用の動的アスペクト指向システムである。実行にかかる時間を計測したい範囲を変更するために利用者は実行時に動的にアスペクトを入れたり外したりすることができる。そのため、KLAS の利用者はアスペクトを変更する度に OS の再起動をする必要がない。この機能のために、利用者は再起動してから調べたいおかしな挙動が再び起こるようになるまで待つ必要が無くなり、性能劣化を引き起こしている点を効率良く探すことができる。つまり、利用者は調べたい場所を見つけてすぐにその部分の調査を開始できる。

KLAS はユーザーランドで実行された KLAS のコマンドを通して利用者よりアスペクトを受け取る。そして、KLAS は受け取ったアスペクトを実行中のカーネルに対して動的に適用する。C 言語のプログラムはコンパイル時に構造体名などのシンボルの情報が消えてしまう。KLAS では改造した gcc を用いることでこのような消えてしまう情報を保存する。そして、これらの情報を利用することで KLAS の利用者はソースコードレベルの視点に基づいてコード挿入が可能となる。

KLAS を使うことで、利用者は構造体のメンバーへのアクセスをポイントカット (KLAS を用いてプロファイリングのためのコードを挿入する点) として抽出することができる。既に述べたように、特定の構造体メンバにある関数ポインタがアクセスされたときにアドバイス (KLAS の挿入したプロファイリングのためのコード) が実行されるというのは重要である。たとえば、mbuf 構造体をアクセスしているような関数の中での時間を測定することが容易にできるので、このような機能は我々が性能低下を引き起こしている点を見つける時の大きな助けとなる。

KLAS のアスペクト記述は XML で行う。KLAS のアスペクトの記述例を図 1 に示す。この例では m_ext 構造体内の ext_free メンバへのアクセスが行われる点をポイントカットしている。そして、ext_free は関数ポインタであるのでそのメンバアクセスは関数呼び出しとなる。このアスペクトのアドバイスはメンバアクセスがされた際に現在の時刻と関数の引数を表示するようになっている。なお、KLAS は$eip、$ebp $esp という特殊な変数をアドバイスの中で使うことができ、それぞれ eip、ebp、esp レジスタの値を表している。

3.2 実　　装

KLAS は指定した実行点に実行が到達した際にアド

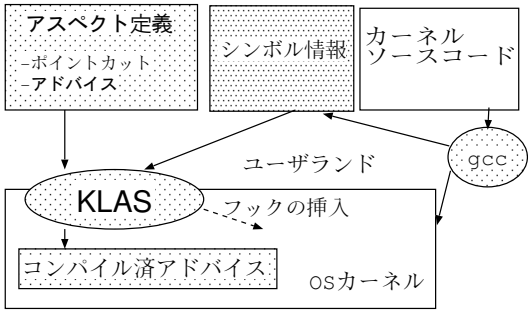

図 2 KLAS の実装

バイスを実行するために OS カーネルにフックを挿入する。KLAS は利用者に与えられたポイントカットに基づく実行点に動的にフックを挿入するためフックのためのオーバーヘッドは最小になる。そして、アスペクトがアンウィーブ (対象としているプログラムからアスペクトを外す処理) された場合には挿入されたフックも実行中の OS カーネルから取り除かれる。KLAS の対象とするアプリケーションはプロファイリングであるため、オーバーヘッドをできるだけ小さくするのは重要である。もし、アスペクトを使うオーバーヘッドが無視できないほど大きければ、測定の影響による値が大きくなり、性能を劣化させている箇所を見つけるという本来の目的を達成するのは難しくなる。

3.2.1 KLAS 用 C コンパイラ

構造体のメンバへのアクセスをポイントカットとして抽出できるという機能は KLAS 独自のものである。これを実現するために KLAS ではコンパイルされたバイナリの持つシンボルテーブルを拡張している。このような粒度の細かいポイントカットにより利用者は OS カーネル内部の性能が劣化している原因となる箇所の特定を効率よく行うことができる。利用者はソースコードのレベルで調べたい構造体のメンバへのアクセスをポイントカットとして指定し、KLAS はそのメンバアクセスに対応する機械語にフックを挿入するために拡張したシンボルテーブルを用いる (図 2)。

KLAS の利用者は我々が拡張した GNU C コンパイラ (gcc) に-g というデバッグオプションを付けてカーネルをコンパイルしなくてはならない。そして、コンパイル中に我々のコンパイラは構造体のメンバへのアクセスがなされる場所のファイル名および行番号とその構造体名、メンバ名の対応づけを調べ、その対応づけをファイルへと保存する。通常のコンパイラでは-g オプションをつけていた場合でもこれらの情報はコンパイル後には失われる。GNU C コンパイラでは構文解析木が作られた後に構造体名やメンバ名の情報が

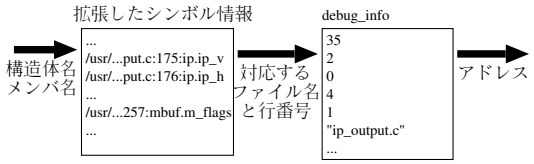

図 3 構造体メンバへのアクセスのアドレス解決

失われ、これらのシンボル名の代わりに整数値の ID 番号を用いてシンボルを表すようになる。このため、我々のコンパイラでは名前と ID の間の対応づけを保持しておき、ファイルへの出力で構造体名やメンバ名が保存されるようにしている。

3.2.2 アドレスリゾルバ

OS カーネルの実行中に利用者から新しいアスペクトのウィーブ (カーネル内部へのプロファイリングのためのコードの挿入) が要求されると、KLAS は OS コンパイル時に我々が改造したコンパイラが生成したシンボル情報を参照する。KLAS は与えられたポイントカットに対応する機械語のアドレスをシンボル情報を用いることで見付ける。関数のアドレスを調べるには、KLAS は nm コマンドを実行して通常のシンボルテーブルを参照する。そして、その結果からアドレスを得る。

また、構造体のメンバにアクセスしている箇所のアドレスを調べるには、KLAS は次の 3 つのステップで行う (図 3)。まず、KLAS は我々が拡張したコンパイラの出力したシンボル情報を用いて、ポイントカットで与えられた構造体のメンバへのアクセスに対応するファイル名と行番号を調べる。そして、KLAS は通常のシンボルテーブルに含まれるデバッグ情報を用いてファイル名から対応するコンパイレーションユニットを特定する。このコンパイレーションユニットとは OS カーネルを構成するオブジェクトファイルのことで、通常は 1 つの C 言語のファイルは 1 つのオブジェクトファイルにコンパイルされるためファイル名とコンパイレーションユニットは 1 対 1 で対応する。最後に、コンパイレーションユニットに対応する DWARF2 形式の **debug_line** の情報を用いることで行番号から対応するアドレスを求める。なお、この **debug_line** の情報も通常のシンボルテーブルに含まれている。

以上のような機構で行うので KLAS はジョインポイント (フックを挿入できる箇所) の含まれる行の最初の機械語のアドレスしか調べることができず、KLAS はジョインポイントに対応する正確なアドレスにフックを挿入することはできない。しかし、OS カーネルの性能を劣化させる原因となる箇所を調べる用途ではこ

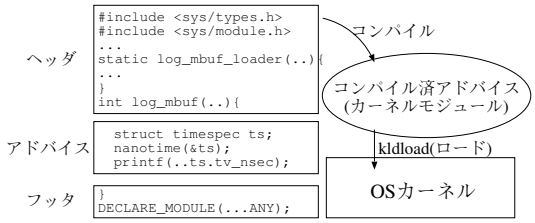

図 4 アドバイスのロード方法

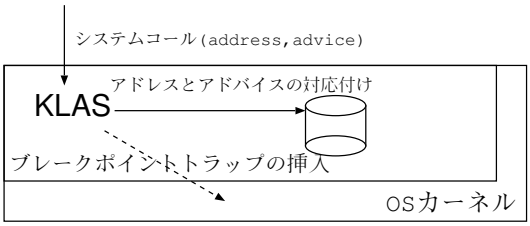

図 5 KLAS のカーネル内部での実装

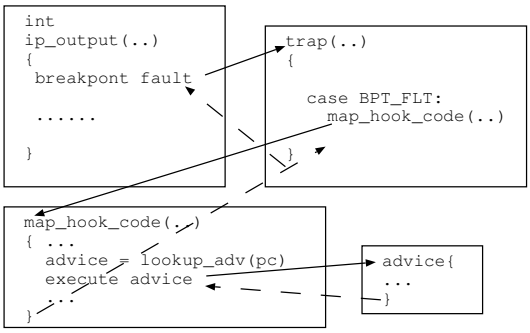

図 6 KLAS におけるアドバイスの実行

れはそれほど大きな問題にはならないと我々は考えている。また、我々の拡張した GNU C コンパイラは構造体のメンバへのアクセスとそれが行われているファイル名、行番号の対応づけを作成するだけである。したがって、上記の方法を使ってアドレスを求めるのには利用者が利用したいコンパイラを使ってカーネルをコンパイルできるという利点がある。

3.2.3 アドバイスローダ

KLAS は GNU C コンパイラ (gcc) を用いてアドバイスの本体をコンパイルし、**kldload** コマンドを用いてコンパイルされたアドバイスの本体をカーネルにロードする (図 4)。まず、KLAS は XML で書かれたアスペクト定義を解析した後にアドバイスの本体を取り出し、それにヘッダ部分とフッタ部分を取り付け、カーネルモジュールとしてのソースコードの体裁を整える。そして、そのソースコードを gcc でコンパイルする。それから、コンパイルされたバイナリは kldload コマンドをによりカーネル内に読み込まれる。なお、カーネル内で利用できる関数のみを使った C 言語のプログラムであればどのようなコードでもアドバイスの本体として書くことができる。

3.2.4 ジョインポイントでのアドバイス実行

3.2.3 節の方法でロードされたアドバイスは動的なウィーブのためのシステムコールが発行された際にカーネルにウィーブされる。KLAS は 3.2.2 節の方法にて特定したジョインポイントに対応する機械語をブレークポイントトラップ命令で置き換える (図 5)。このブレークポイントトラップ命令は KLAS ではフックとして使われている。以上の置き換えは OS カーネルの動作中に行われる。

x86 アーキテクチャにおけるブレークポイントトラップ命令の長さは 1 バイトであり、どのような機械語でもこの命令で置き換えることができる。そして、アスペクトがアンウィーブされたときは挿入したブレークポイントトラップ命令は元の機械語に戻される。なお、jmp 命令は 5 バイト命令であるため、jmp 命令をフックとして用いることはできない。なぜなら、もし基本ブロック☆の最後にあった 1 バイト命令が jmp 命令で置き換えられたとしたら、基本ブロックの次にあった命令列の最初の部分が jmp 命令の一部により上書きされて、そこにどのような意味を持つか分からない命令が入ることになるからである。この状況はシステムダウンを引き起こすおそれがある。しかし、我々の実験では jmp 命令を使った実装はブレークポイントトラップ命令を使った実装の 25 倍速いことが確認されている。そこで、我々はジョインポイントが基本ブロックに納まる場合に限り jmp 命令により実装することを検討している。

KLAS によって設置されたブレークポイントトラップ命令をスレッドが実行した場合にはブレークポイントトラップが発生する (図 6)。トラップハンドラの中で我々が実装した map_hook_code 関数が実行される。この関数は発生したブレークポイントトラップの箇所のジョインポイントに対応するアドバイスの本体を見付け、それを実行する。アドバイスの実行後に、この関数はブレークポイントトラップ命令で置き換えられた元の命令を実行する。KLAS はこれを FreeBSD のカーネルデバッガである DDB と同じ方法で実現している。

☆ プログラムの実行フローが分岐したり他のフローからジャンプなどで入って来たりすることが無いブロック

4. プロトタイプ実装の性能

我々の手法の実行時オーバーヘッドがどの程度になるのか調べるため、フックを扱う箇所のプロトタイプ実装を作成し、システムのオーバーヘッドを測定した。このプロトタイプ実装は FreeBSD のカーネルデバッガである DDB を改造することで作成した。プロトタイプ実装の処理の流れは次の通りである。まず、ブレークポイントトラップにより制御が DDB に移ったところで我々のシステムが設置したブレークポイントトラップであるか調べる。そして、もし我々のシステムが設置したブレークポイントトラップであった場合はアドバイスを実行したのちに DDB のプロンプトを表示せずにデバッガで continue を行った時と同じ動作を行う。

実験ではアドバイスの内容として空のものを与え、システムを実行したオーバーヘッドを測りやすいようにした。なお、プロトタイプを作成し、そのオーバーヘッドを測定した環境は改造した FreeBSD 5.2.1 上で、CPU は AMD Athlon XP 1800+、メモリは 1GB である。

以上の環境で測定した結果は図 7 および図 8 の通りで、挿入するフックの数に対してほぼ $O(n^2)$ で我々のシステムによる負荷が増えていくことがわかった。プロトタイプ実装の各所にかかる時間を測定した結果、ブレークポイントトラップの処理を DDB で行う実装にしたことに原因があることがわかった。KLAS のプロトタイプ実装では DDB によるブレークポイントトラップの処理を利用しているため、アドバイス実行のたびにに全てのフックの挿入、削除が行われている。このように実装することでアドバイスを実行している最中にブレークポイントトラップが発生し、アドバイスの実行が再帰的に行われて止まらなくなるという状況を防げる。しかし、このオーバーヘッドの大きさでは性能が劣化している原因となる箇所を探すのには不向きであるので、jmp による実装も含め、改善策を考える必要がある。

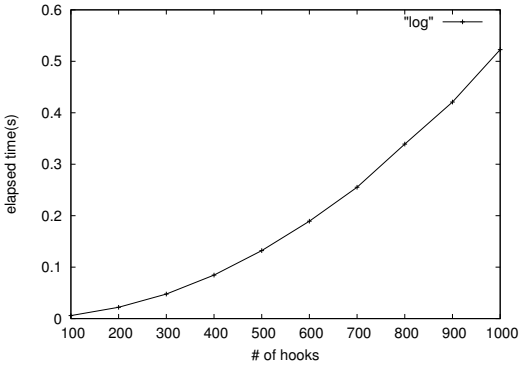

図 7 フックの数と実行時間

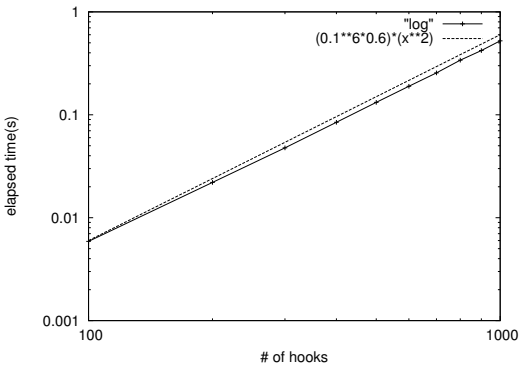

図 8 フックの数と実行時間 (対数表示)

5. 関 連 研 究

5.1 カーネルプロファイラ

Ktr[6] は BSD/OS や FreeBSD に含まれるカーネルプロファイラである。この機能はカーネルコンパイル時に特殊なオプションが渡されることで有効となる。利用者はコンパイルの前にカーネルのソースコード中にロギングのためのコードを手動で書き込む。ロギングのコードはなお、CTRx というマクロを用いて行われている。各々のロギングのコードは ktr_mask と呼ばれる値を参照しており、利用者はこの値を変更することによってマクロが有効か無効かを実行時に変えることができる。しかし、Ktr は KLAS と違い利用者がログを取ることができる箇所を実行中に変更することはできない。もちろん利用者がこれを実現するためにさまざまな場所にこのマクロを挿入し、実行時に必要なところを選ぶということはできる。しかし、このような方法は無視できないほど大きいオーバーヘッドを生むと考えられる。

LKST[12] は Linux 用のカーネルプロファイラである。このシステムはカーネル内部の決められた点において利用者が現在の時間を記録したり、与えられた任意のコードを実行したりすることができる。LKST の問題点は利用者がログを取る場所を指定できないことである。利用者が指定できるのは LKST があらかじめ与えた幾つかの場所である。この制限はカーネルの挙動の詳細を見て、性能が劣化している原因を調べるというのには不向きである。

KernInst[11] および GILK[7] は OS カーネルのコー

ドを実行時に変更することができる。しかし、利用者はどの機械語を別のコードで置き換えるか指定しなくてはならない。そのため、利用者は関数呼び出しや構造体のメンバアクセスといったソースレベルの視点で変更するコードを指定することができない。

5.2 アスペクト指向システム

AspectC++[10] はC++言語用のアスペクト指向システムである。このシステムはカーネルプロファイラに必要な要件の殆どを満たしているが静的なアスペクト指向システムである。よって、利用者はプロファイリングのためのアスペクトを変更する度に再コンパイル、再起動を行わなくてはならない。

μDyner[9] はC言語用の動的アスペクト指向システムである。μDynerの実行時にかかるオーバーヘッドは必ずしも小さくはない。このシステムはコンパイル時にジョインポイントとなりえる全ての場所にフックを実現する特別なコードを入れている。そして、実行時に与えられたアスペクト内のポイントカット記述に基づきそれらのフックが有効な状態になり、制御がそこに到達した際に対応するアドバイスが呼ばれるようにできる。それゆえμDynerを用いてプロファイリングを実現するにはプロファイリングのために調査する全ての箇所にフックを実現するためのコードを入れる必要がある。この数はかなり大きいと考えられるためフックのオーバーヘッドは無視できなくなると考えられる。

TinyC[25] はC言語用の動的アスペクト指向システムの一つである。μDynerと違い、TinyC^{2}は既にコンパイルされたバイナリにフックのコードを実行時に挿入、削除することができる。この機能はDyninst[1]をバックエンドとして利用することで実現されている。フックはポイントカット記述により選ばれた点に対してのみ挿入されるため、TinyC^{2}を利用するオーバーヘッドは非常に小さいものになる。しかしながら、TinyC^{2}においてジョインポイントとなる点は限られている。例えば、関数呼び出しはジョインポイントであるが、構造体のメンバへのアクセスはジョインポイントとはならない。というのも、コンパイル後のバイナリは構造体のメンバアクセスがどの機械語でなされるかというような情報を持たないからである。言うなれば利用者はポイントカット記述にて抽出したい機械語を正確に指定しなくてはならない。

TOSKANA[4] はOSカーネル用の動的アスペクト指向システムである。これはNetBSDの上で動作する。このシステムはアドバイスのカーネル空間へのロードにはKLASと同じアプローチを用いている。そして、フックが分岐命令によって実装されているため、アドバイスの実行時間はKLASのものよりも速いと考えられる。しかし、このシステムは我々のように改造したコンパイラを用いていないため情報が足りず、構造体のメンバへのアクセスをジョインポイントとして抽出することができない。

AspectCはC言語用のアスペクト指向システムである。そして、これを用いた研究によってアスペクト指向がOSのキャッシュの実装において有用であることが示されている[2),3)]。カーネルのキャッシュ処理というのはメモリ操作のモジュールとディスク操作のモジュールにまたがった処理である。そして、これらのモジュールは異なった層に属しているのでアスペクト指向を使わない実装では同じ層にある別のモジュールにまたがった処理を書くのよりも難しいものになる。AspectCではコンパイル時にソースからソースへの変換を行うことでアスペクトのウィーブを実現している。そして、このシステムは動的なウィーブをサポートしていない。また、このシステムは構造体のメンバへのアクセスをジョインポイントとして抽出する機能も備えていない。

PROSE[8] はJava用の初期の動的アスペクト指向システムである。これはJVMDI☆を用いてアスペクトの動的なウィーブを実現している。これはポイントカットによって指定されたジョインポイントに対してブレークポイントを設置してフックを実現している。そして、制御がブレークポイントの何れかに到達した際にはJVM ☆☆は制御をPROSEへと移す。そして、PROSEの中ではジョインポイントに対応するアドバイスを実行する。このやり方はKLASの方法と同じであるが、KLASはC言語で書かれたOSカーネル用のシステムである。

6. ま と め

カーネル内のネットワークの性能を劣化させている箇所を探すのには細かい粒度で時間を測定できるようなカーネルプロファイラが必要である。OSカーネルは多くの多層構造になったモジュールで構成されているため、そのプロファイリングにはアスペクト指向が有用である。さらに、カーネルプロファイラは動的にプロファイリングのためのコードの挿入、削除を行い、再コンパイル、再起動を避けるべきである。なぜなら、コードの挿入、削除の度に再起動を繰り返すようでは

☆ Java Virtual Machine Debugger Interface
☆☆ Java Virtual Machine

性能が劣化しているところを探す効率が悪くなってしまうからである。

本論文では KLAS というアスペクト指向を用いたカーネルプロファイラを提案している。このシステムは構造体のメンバへのアクセスという粒度の細かいジョインポイントを提供しており、利用者はこれを用いて OS の詳しい振舞を知ることができる。KLAS は OS をコンパイルしたときに作られるシンボル情報を拡張している。そして、拡張したシンボル情報には構造体のメンバへのアクセスがされたファイル名、行番号が入っている。KLAS はウィーブ時にこの情報を用いて構造体のメンバへのアクセスをポイントカットとして指定できるようになっている。このようにするのは従来の C コンパイラが Java のコンパイラが作成するのに比べて極めて少量のシンボル情報しか作らないためである。

KLAS のフック挿入部分についてのプロトタイプ実装を行い、その性能を測定した。その結果、フックの数が増えるにしたがって $O(n^2)$ の速さでオーバーヘッドが増加することがわかった。このように、現在の実装での性能は非常に悪いため、たくさんの点にまたがる処理のプロファイリングにはとても使えないと考えている。今後はこの部分の実装を変更し、高速にするようにしていきたいと考えている。また、現在のやり方ではインライン展開やマクロ展開を含めて最適化により元のプログラムの構造が変化して、行そのものがなくなってしまった場合に対応できないため、そのような状況でもトレースできるようにする仕組みが必要であると考えている。

参 考 文 献

1) Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
2) Yvonne Coady and Gregor Kiczales. Back to the future: a retroactive study of aspect evolution in operating system code. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 50–59. ACM Press, 2003.
3) Yvonne Coady, Gregor Kiczales, Michael Feeley, Norman Hutchinson, Joon Suan Ong, and Stephan Gudmundson. Exploring an aspect-oriented approach to os code. In *4th ECOOP Workshop on Object-Orientation and Operating Systems*, june 2001.
4) 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.
5) Gary T. Leavens and Curtis Clifton (eds.). Foal 9,003 proceedings - foundations of aspect-oriented langauges workshop at aosd 2003.
6) Greg Lehey. Improving the freebsd smp implementation. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 155–164. USENIX Association, 2001.
7) David J. Pearce, Paul H. J. Kelly, Tony Field, and Uli Harder. GILK: A dynamic instrumentation tool for the linux kernel. In *Computer Performance Evaluation / TOOLS*, pages 220–226, 2002.
8) 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.
9) Marc Ségura-Devillechaise, Jean-Marc Menaud, Gilles Muller, and Julia L. Lawall. Web cache prefetching as an aspect: towards a dynamic-weaving based solution. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 110–119. ACM Press, 2003.
10) Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002.
11) Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Operating Systems Design and Implementation*, pages 117–130, 1999.
12) 畑崎 恵介, 中村 哲人, and 芹沢 一. 稼働中システムのデバッグを考慮した OS デバッグ機能. 情報処理学会研究報告, 社団法人 情報処理学会, aug 2003.