Vol. 48 No. SIG 0(PRO 34) 情報処理学会論文誌：プログラミング 2007

OS カーネル用アスペクト指向システム KLASY

柳澤佳里[†] 光来健一[†]
千葉滋[†] 石川零[†]

本稿では C 言語で書かれた OS カーネル用の動的アスペクト指向システム KLASY を提案する。他の類似のシステムと異なり、KLASY は関数実行だけでなく構造体メンバへのアクセスを選択（ポイントカット）して、コード（アドバイス）を実行させることが可能である。この機能により開発者がアスペクト指向を用いて OS カーネルをプロファイリングしたり、デバッグしたりすることが容易になる。構造体メンバアクセスをポイントカットできるようにするため、KLASY は我々が改造した C コンパイラを用いて OS カーネルをコンパイルし、拡張シンボル情報を出力する。拡張シンボル情報を用いることで、KLASY のウィーバは OS 実行中に構造体メンバアクセスが行われる箇所のアドレスを探し、アドバイスを実行させるようにすることができる。その際に、ローカル変数などの実行時コンテキストも利用することができる。我々は KLASY を GNU C コンパイラを改造して Linux 上に実装し、実験を行った。その結果、本システムのオーバーヘッドは小さいことがわかった。また、KLASY を用いてシステムの性能劣化の原因を調査したケーススタディを通して、本システムが現実の問題に利用可能であるとわかった。

KLASY: System for Source-based Binary-level Dynamic Weaving

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

In this paper, we propose KLASY, which is a dynamic aspect-oriented system for OS kernels written in C language. Instead of other similar systems, KLASY enables developers to select not only executions of functions but also accesses to structure-members as pointcuts. This feature helps developers to profile and debug OS kernels. To let developers to select accesses to members of structures, we modified a C compiler to generate extra symbol information. The extra symbol information enables a weaver of KLASY to investigate an address where the member of the structure is accessed at run-time. The weaver inserts a hook to make kernel execute an advice when a thread reaches the member access. At that time, execution context can be obtained in an advice body. We implemented KLASY by modifying the GNU C compiler on the Linux to do some experiments. The results of experiments has shown that KLASY has little overhead. Our case studies to investigate performance bottlenecks of Linux kernel has also shown that KLASY is available for real problems.

1. はじめに

オペレーティングシステム（OS）カーネルの性能向上は現在でも重要な課題である。例えば、Linux や FreeBSD の開発者は今でも新しいスケジューラーを開発し続けている。[14),16),19),23)] このような際、動的アスペクト指向システムは OS の性能のボトルネックを調べるのに役立つ。なぜなら、動的アスペクトシステムを用いると、実行時にプロファイリングのためのコードを挿入したり、そのコードを削除したりすることが出来るため、実行中のカーネルを止めることなく調査を続けることが出来るからである。

OS カーネルは C 言語で開発されていることが多く、構造体を多用している。構造体はデータを関連する関数間で受け渡したり、関数ポインタを用いたポリモルフィズムを実現したりするために使われる。そのため、データの流れを調べたり、ポリモルフィズムが使われているときの制御の流れを調べる必要がある。動的アスペクト指向システムは構造体のメンバへのアクセスを選択（ポイントカット）し、プロファイリングのためのコード（アドバイス）を実行できることが望ましい。

また、プロファイリングを行うには、プロファイリングコードを実行する各点で対象の構造体の内容やロー

† 東京工業大学 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

カル変数などの実行時コンテキストを取得できるようにする必要がある。実行時コンテキストが取得できれば、実行中の変数の値を調べることで、ボトルネックをより詳しく調べる可能である。これに加え、実行時コンテキスト情報を利用することで、調べる必要のない情報をログに残すことがなくなるため、ログを保存する領域の節約にも繋がる。

以上の点を満たす動的アスペクト指向システムとして本論文では *KLASY*（Kernel-Level Aspect-oriented SYstem）を提案する。KLASY は構造体メンバへのアクセスをポイントカットする機構を提供しており、OS カーネル内のサブシステムを簡単な記述で網羅的に調査することが可能となっている。また、KLASY はそれらのメンバアクセスが行われた点での実行時コンテキストを取得できるため、ローカル変数などの情報に基づいた綿密な調査を可能としている。

以下、2 章にて既存のアスペクト指向を用いたカーネルプロファイリングとその問題点について論じ、3 章にて KLASY を提案する。4 章では KLASY のオーバヘッドに関する実験とケーススタディについて論じる。5 章では関連研究について論じ、6 章で本論文をまとめる。

2. アスペクト指向を用いたカーネルプロファイリング

アスペクト指向プログラミング（AOP）は、プロファイリングのためのコードを OS カーネルのソースコードから分離したモジュールとして記述することを可能にする。このモジュールは**アスペクト**と呼ばれ、**ポイントカット**と**アドバイス**からなる。ポイントカットはいくつかの述語の集まりであり、それらの述語に一致した箇所が選択される。ポイントカットで選択可能な箇所は**ジョインポイント**と呼ばれる。アドバイスは関数のような言語機構であり、プログラムの実行がポイントカットで選択された箇所に到達したときに呼び出される。対象プログラムに対してポイントカットで指定された箇所とアドバイスを結びつける操作は**ウィーブ**と呼ばれる。

プロファイリングやロギングは AOP が役立つ例であるが、既存の C 言語用の AOP システムは OS カーネルのプロファイリングには不十分である。C 言語用の AOP システムはプロファイリングを行うアスペクトを動作中の OS カーネルを再起動することなく適用したり取り外したりできなくてはならない。C 言語用の動的 AOP システムはいくつか提案されているが、それらのシステムは非常に限定された種類のジョインポイントしか扱うことができない。例えば、TOSKANA[9] は関数実行しかジョインポイントとして選択することができない。Arachne[8] は関数の呼び出しだけでなく、グローバル変数や任意のメモリブロックへのアクセスも選択できるが、これらのシステムのいずれも構造体のメンバへのアクセスを選択することはできない。

OS カーネルでは、関数間でまとまったデータをやりとりするために頻繁に構造体が用いられる。データの流れを調べるには、カーネル開発者はポイントカットで構造体のメンバへのアクセスを選択できることが望ましい。関数の実行だけしかジョインポイントとして選択できないなら、関数間でやりとりされるデータの流れを追うのは容易なことではない。

さらに、構造体は OS カーネルにてポリモルフィズムを実現するためにも用いられる。この場合、C++ や Java におけるクラスの代わりとして用いられる。例えば、構造体のメンバのいくつかは関数ポインタになっており、呼び出し元は実際に呼び出される関数を意識することなく、同一のインタフェースで下位のサブシステムを扱えるようになっている。ネットワーク I/O システムや仮想ファイルシステム、デバイスドライバなどの機能はこのようにして実装されている。そのため、カーネル開発者がこの種の関数呼び出しを調査するために、構造体内の関数ポインタが格納されているメンバへのアクセスをジョインポイントとして選択できれば便利である。

既存の C 言語用の動的 AOP システムのもう一つの問題は実行時のコンテキストを十分に利用できないことである。一般に AOP システムにはジョインポイントにおけるコンテキスト情報をアドバイスに渡すための仕組みがある。例えば、関数実行がジョインポイントの場合は関数の引数をアドバイスに渡すことができる。Arachne ではこれに加えて関数の返り値やグローバル変数にもアクセスすることができる。しかし、既存の C 言語用動的 AOP システムではジョインポイントで有効なローカル変数をアドバイス本体で参照することはできない。ローカル変数をアクセスできないことはモジュール化の観点から考えると適切なデザインではあるが、プロファイリングでは必要となることが多い。

3. KLASY: Kernel Level Aspect-oriented System

2 章で述べたように、既存の C 言語用 AOP システムは OS カーネルのプロファイリングやデバッグに使うには不十分である。そこで、我々は C 言語で書

```
<aspect>
  <import>linux/time.h</import>
  <advice>
    <pointcut>
      access(inode.i_uid) AND
      within_function(inode_change_ok) AND
      target(inode_ptr);
    </pointcut>
    <before>
      struct inode *i = inode_ptr;
      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>
```

図 1 KLASY のアスペクト (inode_trace.klasy)

かれた OS カーネルのプロファイリングやデバッグを行うのに適した動的アスペクト指向システム KLASY (Kernel Level Aspect-oriented SYstem) を提案する。他の類似のシステムと異なり、KLASY は構造体のメンバへのアクセスをポイントカットし、ジョインポイントにおける実行時コンテキストにより良くアクセスすることを可能にする。

3.1 KLASY のアスペクト

KLASY のアスペクトは XML のタグで囲まれた C 言語で記述する。図 1 は KLASY のアスペクトの例である。 このアスペクトがカーネルにウィーブされると、inode_change_ok() 関数の中で inode 構造体の i_uid メンバがアクセスされたときにログメッセージが出力される。なお、KLASY はウィーブしたアスペクトを実行中の OS カーネルから取り除く（アンウィーブ）こともできる。

アスペクトは aspect タグで囲まれており、import タグはアドバイス本体をコンパイルするのに必要なヘッダーファイルを指定するのに使われる。アドバイス本体は C 言語で書かれており、before あるいは after タグで囲まれている。それぞれジョインポイントとの前または後で実行される。KLASY は現在 AspectJ でいう around アドバイスをサポートしていなが、これはプロファイリングやデバッグ目的に KLASY を使う限り致命的な問題ではない。デバッグ用途では around が利用可能であれば、動作中の OS にパッチをあてて挙動の変化を見るといったこともできるが、利用可能でなくても致命的ではないと考えられる。実際、一般的なデバッガにはこのような機能はない。なお、アドバイス本体は XML のタグの中に入っているため、& や> のようにいくつかの文字はエスケープしなければならない。

アドバイス本体の前には pointcut タグで囲まれたポイントカットの定義を記述する。KLASY は現在、access、execution などのポイントカット記述子を提供している。access ポイントカット記述子は構造体名とメンバ名を繋いで指定することで構造体のメンバへの読み書きをジョインポイントとして選択する。execution ポイントカット記述子は関数の実行を選択することができる。これらのポイントカットでは%をワイルドカードとして用いることができる。また、within_file ポイントカット記述子や within_function ポイントカット記述子を用いることで、選択するジョインポイントを指定した関数やファイル内に制限することができる。複数のポイントカットを使うときは AND 演算子や OR 演算子でつないで使うことができる。例えば、within_function と access ポイントカットが AND で結ばれていたら、選択されるジョインポイントは within_function と access の両方の条件を満たすものとなる。KLASY には within_file に加え、選択されるジョインポイントの範囲を指定したファイル内に制限することができる within_file ポイントカット記述子がある。

構造体のメンバへのアクセスを access ポイントカットで選択した場合には、target ポイントカット記述子と local_var ポイントカット記述子を用いることができる。target ポイントカットは access ポイントカットで選択されたメンバアクセスが行われる構造体変数への参照をアドバイスに渡すために用いられる。図 1 では、inode 構造体変数へのポインタが inode_ptr 変数に結びつけられ、アドバイス本体の中で利用できるようになっている。なお、inode_ptr の型は void* である。また、local_var ポイントカットを使うことで、ジョインポイントでのローカル変数を取得することができる。通常、local_var ポイントカットを使うときは within_function ポイントカットと AND 演算子で繋いで使い、選択されるジョインポイントの範囲を関数内に制限する。例えば、BSD プロセスアカウンティング情報をファイルに書き込む箇所（kernel/acct.c の do_acct_process 関数）にて実際に書き込むアカウンティング情報を取得するには

```
access(file_operations.write) AND
within_function(do_acct_process) AND
local_var(ac, ac_ptr)
```

のようなポイントカットを記述する。これは file_operations 構造体の write メンバに格納された関数ポインタを呼んでいる箇所を選択し、ローカル変数 ac の値として与えられるアカウンティング情報を local_var ポイントカット記述子を用いて取得している。local_var ポイントカット記述子は第 1 引数で指定された変数 ac の値へのポインタを第 2 引数で指定した ac_ptr という

変数名でアドバイス内で利用することを示している。

関数実行を execution ポイントカット記述子で選択したときは、argument ポイントカット記述子を用いて、選択した関数の引数への参照をアドバイス本体に渡すことができる。argument ポイントカット記述子の使い方は local_var ポイントカット記述子と同様である。なお、target ポイントカット記述子及び local_var ポイントカット記述子は access ポイントカットとのみ用いられ、argument ポイントカット記述子は execution ポイントカットとのみ用いられる。さらに、access ポイントカットと execution ポイントカットはその性質上 AND 演算子で結ぶことが出来ない。これら、誤ったポイントカット記述を行った場合はエラーとなり、アスペクトはウィーブされない。

3.2 Source-based binary-level dynamic weaving

KLASY はアスペクトをウィーブするのに *source-based binary-level dynamic weaving* という手法を用いる。この手法は、コンパイル時に取得したソースレベルのシンボル情報を用いて、実行中の OS カーネルバイナリにアスペクトを動的にウィーブする手法である。この手法により、構造体メンバアクセスの箇所のアドレスおよび、実行時コンテキストを取得できる。KLASY におけるアスペクトのウィーブは図 2 のように行われる。まず、OS カーネルのソースコードを我々が拡張した C コンパイラでコンパイルし、拡張シンボル情報を得る。次に、アスペクトを KLASY のアスペクトコンパイラを使ってコンパイルする。アスペクト内のアドバイス本体はローダブルカーネルモジュールに変換されてカーネルにロードされる。KLASY は拡張シンボル情報を元にポイントカットで選択されたジョインポイントのメモリアドレスを調べ、それらのメモリアドレスにフックを挿入する。フックとはそのメモリアドレスに制御が来たところで、アドバイス本体を呼び出すための機構である。

3.2.1 拡張シンボル情報の取得

access ポイントカットに対応するため、我々は gcc のパーザを拡張し、構造体のメンバアクセスが行われている箇所が記録されるようにした。gcc では lineno や input_filename といったグローバル変数にパーズ中の行番号やファイル名が格納されている。拡張したコンパイラはメンバアクセスが見つかったところで、その構造体名とメンバ名、および、そのメンバアクセスが行われているファイル名と行番号を記録する。

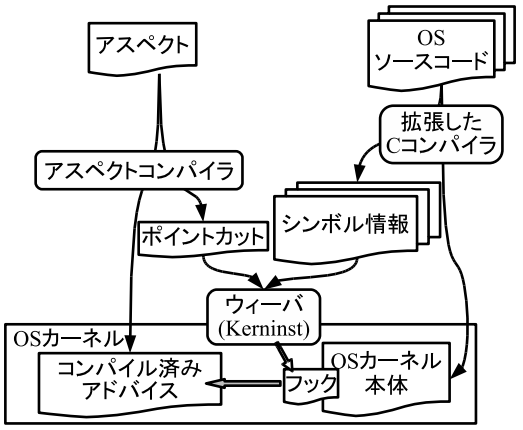


図 2 KLASY におけるアスペクトのウィーブ

メンバアクセスのあるジョインポイント☆に対応した機械語のあるメモリアドレスを見つけるため、KLASY は gcc のデバッグ情報を用いる。gcc のデバッグ情報はデバッグオプション（-g）をつけてコンパイルすることで得られる。まず、KLASY は access ポイントカットで指定された構造体名とメンバ名からメンバアクセスが行われるファイル名と行番号を得る。次に、デバッグ情報を用いて、ファイル名と行番号から対応する機械語のあるメモリアドレスを得る。.c ファイルにメンバアクセスが出現する場合は、対応するオブジェクトファイルのデバッグ情報を調べればよい。

一方、ヘッダファイルで定義されたマクロにメンバアクセスが出現する場合は、対応するオブジェクトファイルが存在しないため、全てのオブジェクトファイルのデバッグ情報を調べる必要がある。ヘッダファイルで定義されたマクロで行われるメンバアクセスは様々な.c ファイルで使用される可能性があるためである。OS カーネルに含まれるオブジェクトファイルの数は膨大であり、これら全てのデバッグ情報を探索すると非常に時間がかかる。そこで、ヘッダファイルで定義されるマクロで行われるメンバアクセスに対しては、KLASY はマクロが使用される.c ファイルの行番号情報を記録する。これにより、KLASY は各.c ファイルに対応する 1 つのオブジェクトファイルのデバッグ情報を調べるだけで済む。

しかし、従来の gcc が生成するデバッグ情報はこのような目的のためには不十分であった。ヘッダファイルで定義されたマクロを.c ファイルで使用していると、従来の gcc はマクロが定義されているヘッダファ

☆ 正確にはジョインポイントシャドウ[15] だが、説明を簡単にするためにジョインポイントと書く。

イルの行番号情報しかデバッグ情報に記録しない。このため、マクロが定義されているヘッダファイルではなく、マクロを使用している.c ファイルの行番号情報を記録するようにすると、機械語のあるメモリアドレスを得ることができない。また、.c ファイルでマクロを使用している行にポイントカットしたいメンバアクセスがある場合、そのメンバアクセスに対応する機械語のあるメモリアドレスを得ることができない。さらに、gcc が最適化を行うと多くの行番号情報が失われてしまい、ポイントカットすることができなくなる。この問題は通常、-Os オプションで最適化される OS カーネルにとっては致命的である。

そこで、我々はデバッグ情報になるべく多くの行番号情報を残すように、gcc および gas（GNU アセンブラ）を改造した。gcc は初期段階ではヘッダファイルおよび.c ファイルのすべての行番号情報を残すが、RTL（レジスタ遷移言語）生成器により 1 つのメモリアドレスに結びつけられた行番号情報は 1 つを残して削除される。さらに、RTL 最適化器は最適化に関連したコードに対応する行番号情報の多くを削除する。gas は 1 つのメモリアドレスに結びつけられた行番号情報がたとえ複数あっても、そのうち 1 つについてしかデバッグ情報を生成しない。我々は 1 つのメモリアドレスに複数の行番号情報が結びつけられるように gcc および gas を改造した。また、最適化が行われてもなるべく行番号情報を残すようにした。これにより、マクロが使用されている行に関しても、.c ファイルの行番号情報を利用することができるようになった。また、最適化が行われた場合でも、プロファイリングには問題ない程度の誤差で機械語のアドレスを得ることができるようになった。

3.2.2 メンバアクセス箇所のアドレスの取得

KLASY では拡張した gcc により生成されたシンボル情報を調べることで、ジョインポイントに対応するメモリアドレスを得る。まず、KLASY は拡張シンボル情報からポイントカットで選択されたジョインポイントに対応するファイル名、行番号を調べる。次に、デバッグ情報を元に行番号に対応した機械語のあるメモリアドレスを調べる。 この際に、KLASY はまずコンパイルされたバイナリ中の.debug_info セクションを読み、ファイル名に対応するオブジェクトファイル単位の情報を得る。次に、.debug_line セクションを読んで、行番号に対応するメモリアドレスを得る。

デバッグ情報からはメンバアクセスを行っている正確なアドレスが得られないため、KLASY は行単位でフックを挿入する。そのため、たとえば before アドバイスはジョインポイントの直前ではなく、ジョインポイントの含まれる行の直前で実行される。行番号から得られたメモリアドレスからバイナリを解析することで正確なアドレスを見つけられる可能性はあるが、現状では対応していない。行単位という粒度はソースレベルのデバッガや典型的なプロファイラと同等の粒度であるため、OS カーネルのデバッグやプロファイリングに使う上ではこのような仕様は妥当だと考えられる。

3.2.3 フックの挿入

ジョインポイントのあるメモリアドレスが見つけたら、KLASY は Kerninst[20] を用いてそこにフックを挿入する。アドバイス本体はローダブルカーネルモジュール内部の C 言語の関数に変換されており、フックはジョインポイントのメモリアドレスを引数とする関数呼び出しを行うコードとなっている。フックを挿入するとき、Kerninst はそのアドレスにある元の命令をジャンプ命令に置き換える。そして、ジャンプ命令からはカーネルのメモリ内に動的に配置された KLASY のコードへと処理が移る。ジャンプ命令で上書きした場所にあった命令はこのコードの実行後に実行される。置き換える命令がジャンプ命令 (5 バイト) のサイズよりも小さかった場合、Kerninst は 1 バイト命令であるブレークポイントトラップをそこに配置する。そして制御がその命令に至ったところでトラップハンドラが呼び出されフックコードを実行した後にブレークポイントトラップで置き換えた元の命令を実行する。ブレークポイントトラップはソフトウェア割り込みを発生させ、ジャンプ命令に比べて処理に大幅に時間がかかる。このため、Kerninst はジャンプ命令で元の命令を置き換えられない場合にのみブレークポイントトラップを用いる。

同じジョインポイントに複数のアドバイス本体をウィーブできるようにするため、KLASY はアドバイス本体を一つずつ順番に実行するトランポリン関数を作る。フックからは実際にはアドバイス本体ではなくこの関数が呼び出される。そして、トランポリン関数は実行時コンテキストの取得に必要な処理をした後に、そのジョインポイントにウィーブされているアドバイス本体を 1 つずつ呼び出す。このような実装になっているのは Kerninst のみで複数のアドバイス本体を呼び出すことができないからである。

3.2.4 実行時コンテキストの取得

local_var ポイントカット、target ポイントカット、argument ポイントカットの 3 つのポイントカット記述子はジョインポイントにおけるローカル変数やメン

バアクセスが行われる構造体、関数の引数への参照をアドバイス本体に渡すのに用いられる。このための処理は Kerninst の入れたフックとアドバイス本体の橋渡しをするトランポリン関数が行う。フックからトランポリン関数を呼び出すと、トランポリン関数では実行時コンテキストから必要な値を取得し、それをアドバイス本体から生成された関数に引数として渡す。

ローカル変数や関数の引数を得るため、KLASY はまずカーネルバイナリの中の.debug_info セクションを読み、変数に割り当てられているレジスタの番号やメモリアドレスを調べる。このとき、ローカル変数の格納場所を正確に知るために-fno-omit-frame-pointer オプションも同時につけてコンパイルし、フレームポインタを残すようにする。変数がレジスタに割り当てられている場合、レジスタの値はトランポリン関数を呼ぶ前に Kerninst によってスタックに保存されるので、トランポリン関数ではスタックを読んでそのレジスタが格納されているメモリアドレス得る。また、変数がスタックフレームに割り当てられている場合は、トランポリン関数はまずスタックフレームポインタが格納されているレジスタの値☆をスタックから読み、それに基づいて値の格納されているメモリアドレスを計算する。

access ポイントカットと target ポイントカットの両方がポイントカットに含まれている場合は、トランポリン関数では access ポイントカットで指示されているメンバアクセスを行っているアドレスを得る。もし構造体がローカル変数である場合は、KLASY は上記の方法でローカル変数のアドレスを得て、それから構造体の参照になるようにアドレスを計算する。例えば、下記のメンバアクセスを考える。

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

これらへのアクセスをポイントカットとして選択した場合、対象となる構造体のアドレスは&inode と inode_ptr の値となる。KLASY の gcc は、これらの値がローカル変数の inode や inode_ptr からどのように計算できるかという情報を拡張シンボル情報に出力するので、この情報を用いてる構造体のアドレスを得ることができる。

対象となる構造体が格納された変数そのものがローカル変数や関数引数である必要はない。例えば、p というローカル変数があり、 p->thread->fs のようなコードがあったとする。fs へのアクセスがポイントカットとして選択されていたとすると、アドバイス本体に渡すアドレスは p->thread のアドレスである。この場合も、KLASY のコンパイラの作成したシンボル情報から実行時にトランポリン関数でこのアドレスを計算することが可能である。この他、対象となる構造体がインデックスが即値である配列要素の場合もその構造体のアドレスを取得できる。一方、現在の実装では、構造体がインデックス式が変数を含む配列要素である場合や、f()->thread->fs のように構造体を指す式に関数が含まれている場合など、取得したい構造体の格納場所がコンパイル時に決まらない場合は構造体のアドレスを取得することはできない。また、現在の実装では対象となる構造体を指す式の中に、ローカル変数が 2 つ以上含まれていると、その構造体のアドレスを計算することはできない。例えば、配列要素のインデックス式に変数が含まれている場合は配列とインデックスの 2 つの変数を使って格納場所を調べる必要があるので対応していない。このような場合にはカーネルソースコードコンパイル時にエラーを表示し、ローカルコンテキストを得られないことを利用者に通知する。

☆ KLASY は x86 アーキテクチャ上で動作するので ebp レジスタがフレームポインタとして用いられる。

4. 実　　験

我々は KLASY を Linux 2.6.10 カーネル (Fedora Core 2)、Kerinst 2.1.1、gcc 3.3.3 に実装した。開発したコードはコンパイラ部 25380 行、動的ウィーバ部 7728 行で、合計 33661 行である。なお、パッチファイルの行数として gcc に 1588 行、kerninst に 198 行、binutils に 102 行の変更が行われている。開発には FreeBSD 上でのプロトタイプ実装から数えて仕様の決定や設計の時間も含め、3 人年程度かかっている。

本節では KLASY を用いて行った実験の結果について述べる。実験に使ったマシンは、AMD Athlon™ XP 2200+ processor (1.8GHz)、メモリ 1GB、Intel® PRO/1000 ネットワークカードという構成である。

4.1 ユーザランドベンチマーク

4.1.1 KLASY カーネルのオーバーヘッド

現実的な状況での KLASY のオーバヘッドを測るため、我々は UnixBench[21] にてベンチマークを測定した。ベンチマークに含まれるプログラムは、表 1 の通りである。我々は、通常の gcc でコンパイルして静的にリンクしたカーネル（monolithic）、KLASY の gcc でコンパイルして静的にリンクしたカーネル（klasy）、Fedora Core 2 付属のカーネル（normal）の 3 つについて比較を行った。なお、klasy にはアスペクトを

名称	内容
dhry2reg	Dhrystone 2 ベンチマーク
whet	Whetstone ベンチマーク
execl	exec システムコールの性能
pipe	パイプのスループット
context	プロセス間のコンテキストスイッチの性能
file1	256 バイトのファイルコピー
file2	1024 バイトのファイルコピー
file3	4096 バイトのファイルコピー
create	プロセス生成
shell	シェルスクリプト
syscall	システムコール呼び出し

表 1 UnixBench に含まれるベンチマーク

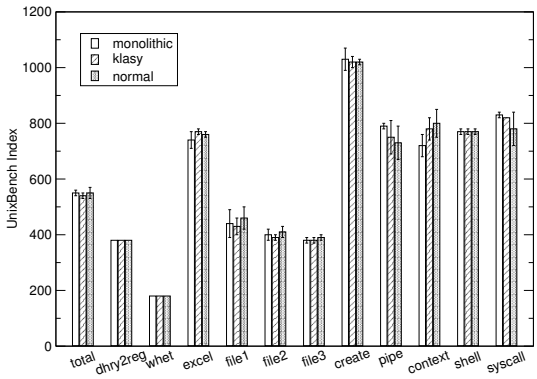


図 3 KLASY のオーバーヘッド

ウィーブしていない。

ベンチマークの結果は図 3 の通りである。ベンチマーク結果は各々のベンチマークが出力した指標であり、数値が大きい方が良い性能を示している。実験結果より、3 つのカーネルの間には大きな性能の違いがないことがわかる。いくつかのベンチマークでは klasy は monolithic よりも少し性能が悪いという結果が出ているが、平均のオーバヘッドは 1%であった。このオーバーヘッドは KLASY のコンパイラがデバッグ情報を出力するために-g オプションおよび-fno-omit-frame-pointer オプションつきでカーネルをコンパイルしていることが原因である。ただし、3 つのカーネルの間のコード配置の違いによるキャッシュヒット率への影響も大きいと考えられる。

4.1.2 アスペクトのオーバーヘッド

次に、4 つのアスペクトをウィーブして CPU をよく使うベンチマークプログラム（dhry2reg, syscall, pipe, execl, context）を動かした。4 つのアスペクトのうち 2 つは runqueue 構造体の nr_switches メンバへのアクセスをポイントカットし、残りの 2 つは task_struct 構造体の state メンバへのアクセスをポイントカットする。メンバアクセスごとにアクセス回数を数えるアスペクトと現在の時刻を記録するアスペクトの 2 種類について測定を行った。nr_switches メンバはコンテキストスイッチがなされたか否かを示し、state は実行中やスリープ中などのプロセスの実行状態を表す。これらのアスペクトをウィーブした場合、nr_switches メンバへのアクセスをポイントカットしているものについては 2 箇所、state メンバへのアクセスをポイントカットしているものについては 50 箇所にフックが入った。

このベンチマークの結果は図 4 の通りである。このベンチマーク結果も数値が大きい方が良い結果を示している。各項目の上にある数値は、カーネル実行中にアドバイスが呼ばれた回数を示している。() で囲まれている数値はジャンプ命令で呼ばれた回数、<>で囲まれている数値はブレークポイントトラップ命令で呼ばれた回数である。ベンチマークはいずれも 38 回行い、平均、分散を計算した。実験結果より、アドバイス本体の呼び出し回数が数千万回以上と極端に多い場合を除き、アスペクトによるオーバーヘッドは許容範囲にあると言える。例えば、(a) の syscall にはアスペクトの有無による性能差があるように見えるが、有意な差があるとは言えない。差がないという帰無仮説をたてて有意水準 10%で t 検定を行い棄却できなかった。同様に検定を行った結果、(a) および (b) の pipe の結果にもアスペクトの有無による有意差がなかった。これに対し、アスペクトの有無で顕著な差を示しているものは context の (b) である。これは (b) ではアドバイス呼び出しのほとんどがブレークポイントトラップで行われているのが原因だと考えられる。実際、我々が行った予備実験ではアドバイスの呼び出しにかかる時間はジャンプ命令によるフックでは約 20 ナノ秒、ブレークポイントトラップ命令によるフックでは約 200 ナノ秒かかることがわかっている。

4.2 実アプリケーションによるケーススタディ

次に KLASY の特徴である access および target ポイントカットを活用したプロファイリング例を示す。

4.2.1 ネットワークトレーシング

KLASY を開発した理由の一つは過負荷時におけるネットワーク I/O サブシステムのボトルネックを発見することであった。そこで我々は、scp コマンドでバルクデータをリモートホストから受信しているときの経過時間をネットワーク I/O サブシステムのいくつかの点で計測した。その結果、ボトルネックの主な原因はプロセススケジューリングによるものだとわかった。

図 5 はこの実験に用いたアスペクトである。このアスペクトは sk_buff 構造体の全てのメンバへのアクセス

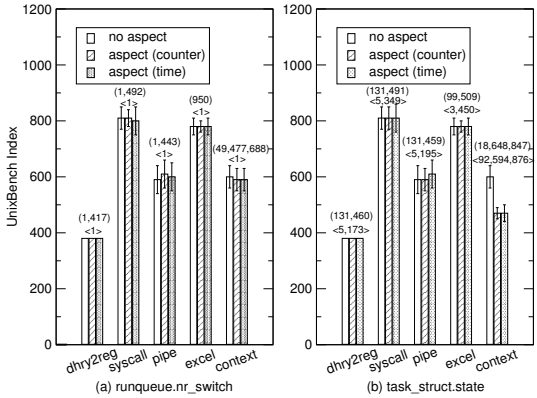


図 4 アスペクトによるオーバーヘッド

```
<aspect>
            :
  <import>local.h</import>
  <advice>
    <pointcut>
      access(sk_buff.%) AND target(arg0)
    </pointcut>
    <before>
      struct sk_buff *skb = 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>
```

図 5 ネットワーク I/O をトレースするアスペクト

をポイントカットして、arg0 にその構造体への参照を格納する。local.h は我々が書いたヘッダファイルであり、.c ファイルに書かれたデータ構造の定義をコピーしたものである。アドバイスでは target で取得した構造体へのポインタを sk_buff 型のポインタ変数に代入している。この構造体の情報に基づき、プロトコルが ARP でない場合に限り、現在の時刻とプログラムカウンタの値（pc）を記録する。DO_RDTSC というのは KLASY で定義されているマクロで、CPU の現在のタイムスタンプカウンタの値を得る。STORE_DATA は KLASY が定義するもう一つのマクロで、カーネル内のメモリに引数で指定した値を保存する。このデータは後でユーザプログラムから読み出すことができる。

このアスペクトのウィーブでは、2494 箇所へのフックを挿入できたが、297 箇所へのフック挿入は失敗した。このうち、70 箇所ではジョインポイントのアドレスを調べられないために失敗していて、227 箇所では target で指定した値が得られないために失敗していた。失敗した原因を詳しく調べてみた結果、前者の失敗の原因は複数行にわたる文の 2 行目以降にジョインポイントが含まれている場合に起きていることがわかった。更に調査した結果、KLASY はジョインポイントのアドレスを求めるのに行番号を用いているが、gcc は最初の行の行番号しかデバッグ情報に保存しないのが原因だとわかった。そこで、本実験の前に複数行にわたることが多い if 文や while 文については最初の行の行番号をジョインポイントのある行として用いるように変更を行い、測定している。しかし、実験前に対応した if 文や while 文以外の文は多種多様であり、まだ完全には対応できていない。後者の失敗の原因は最適化により target で取得する変数が失われてしまった場合に起きていることがわかった。

もし、KLASY を使って Linux カーネルの機能を拡張する場合はこのような失敗は許容できないだろう。しかし、KLASY の対象とするアプリケーションはプロファイリングやデバッグであり、我々の経験ではこれらの場合はジョインポイントに対してそれほど正確さが要求されないため、この程度の失敗は許容範囲だと考えられる。フックの挿入箇所のアドレスを見つけるのに失敗した場合は KLASY は警告を出すため、KLASY の利用者は KLASY がフックの挿入に失敗したことを知ることができる。

このようにアスペクトをウィーブしたカーネルに対して、scp コマンドをリモートホストで実行した。その結果、各ネットワークパケットについて、対象のホストにあるネットワークデバイスにパケットが到着したところから、ユーザプログラムが受け取るまで、ネットワークサブシステムのいくつもの点で経過時間を計測することができた。表 2 はアスペクトを用いて計測したネットワーク I/O のトレース結果の一部である。紙面の都合により、表では特に差が大きい 2 つのパケットについて、ネットワークから入って来たパケットについてトレースした結果を得た 76 箇所のうち特に重要な 8 箇所を選んで載せている。

対象のホストがパケットを受信した後、まず最初に e1000_main.c の 2773 行目に処理が来る。そして、ここから tcp_input.c の 4355 行目に処理が来るまで、どちらのパケットも 14 から 15 μ秒の時間がかかっている。しかし、ここから datagram.c の 234 行目に処理が移るまでの経過時間は 12 μ秒と 688 μ秒と大きく異なっている。ソースコードを調べると、tcp_rcv_etablished 関数で sk_buff 構造体変数をキューに繋ぎ、skb_copy_data_iovec 関数でそれをキューから取り出していることがわかった。また、skb_copy_data_iovec がユーザプロセスにより実行され

関数名	ファイル名	行番号	パケット 1	パケット 2
e1000_rx_checksum	e1000_main.c	2773	0.00	0.00
netif_receive_skb	dev.c	1638	1.06	1.39
ip_rcv	ip_input.c	367	3.43	4.35
ip_local_deliver	ip_input.c	275	5.56	6.67
tcp_rcv_established	tcp_input.c	4238	11.14	12.62
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

表 2 ネットワーク I/O のトレース結果（抜粋）

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

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

図 6 プロセス切り替えをトレースするためのアスペクト

ていることもわかった。以上のことから、この 2 つパケット間の時間差はプロセススケジューリングが原因で起こっていると推測される。

4.2.2 プロセス切り替えのトレース

我々は以前行った研究[12] で、重いウェブサービスにより過負荷になっているときに軽いウェブサービスの性能が極端に下がる原因が Linux のスレッドスケジューリングにあるということを調べた。この研究では、Tomcat[2] 上で各スレッドが使う CPU のタイムクワンタムを計測したが、この計測のためにカーネルソースコードを変更する必要があった。

KLASY を使うと、このような計測はカーネルのソースコードを変更することなく行うことができる。計測に使ったアスペクトは図 6 の通りである。このアスペクトでは、sched.c ファイルで定義された関数の中にある task_struct 構造体の timestamp メンバへのアクセスをポイントカットしている。このメンバはスケジューラがプロセスを切替えるときなどに値が変更される。そして、アドバイス本体ではプログラムカウンタ、プロセス ID、時刻を記録している。このアスペクトをウィーブして、10 箇所にフックをいれることに成功し、失敗箇所はなかった。

アスペクトをウィーブした後に、我々は Tomcat 上で軽いサービスと重いサービスの両方を動かした。ア

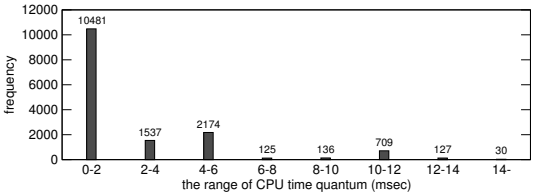


図 7 CPU タイムクワンタムの分布

ドバイス本体にて記録されるログから CPU タイムクワンタムを比較するため、我々はプロセス切り替えに関する箇所（sched.c ファイル中の 2682 行目）のログを集めた。このとき調べたスレッドの CPU タイムクワンタムの分布は図 7 の通りである。この図を見ると、CPU タイムクワンタムには 2 つのピークがあり、二つ目のピークは 10 から 12 の間であることがわかる。この結果は先の研究[12] の過程で得られた結果と同じであった。先の研究と同じ結果を得るには、図 6 に加え、いくつかのアスペクトを書き足さなくてはならないが、紙面の都合により本論文では割愛する。なお、それらのアスペクトも図 6 同様、簡単に書くことができる。

4.2.3 access ポイントカットの必要性

これらのケーススタディで用いたアスペクトは全て access ポイントカット記述子を用いていた。C 言語では、関数を指定することでジョインポイントを選択するよりはメンバアクセスを選択することでジョインポイントを指定する方が簡単な場合が多い。なぜなら、C 言語は C++や Java と異なり、package や class のような関数をグループ化する仕組みがないからである。この節では access ポイントカット記述子がない場合の問題をケーススタディを元に考察し、access ポイントカットの必要性を示す。

4.2.1 節のケーススタディは sk_buff 構造体が Linux のネットワーク I/O サブシステム全体でデータのやり取りに使われているということを利用している。sk_buff のメンバ全てへのアクセスをポイントカットすることで、ネットワーク I/O サブシステムの振舞を把握する

ことができた。また、target ポイントカットで sk_buff のアドレスを記録することで、個々のパケットについて調べることが可能になった。

ここで、従来のシステムにもある execution ポイントカット記述子のみで同じ情報を得ることを考えてみる。この実験でログを取ったポイントは 76 箇所であるが、一つの関数で複数回ログを取っている所もあるため、ログを取った関数は 21 箇所である。このうち 6 箇所は static inline 関数であり、コンパイル後に関数が存在しなくなるので最適化を禁止しなければならない。ログを取った 21 箇所の関数を見つけるにはネットワークからパケットが到着した場合にネットワーク I/O サブシステムのどの関数を実行するかという詳細な知識も必要になる。不要な関数をポイントカットしてしまうと無駄なログを記録している可能性がある。さらに悪いことに、通常 OS カーネルは最適化オプション（-Os）つきでコンパイルされ、インライン修飾子がない関数についてもインライン化による最適化が行われる。そのため、execution ポイントカット記述子で 4.2.1 節の調査をするにはこの種の最適化を抑制した状態で OS カーネルをコンパイルする必要がある。しかし、このような抑制下での調査は現実と大幅にかけはなれており、プロファイリング結果が役に立たないことが多いと考えられる。

また、今回の実験では一つの関数の中で sk_buff 構造体のメンバへのアクセスが複数回出てきている場合はそのつど調査しているため、execution ポイントカット記述子だけで同等のきめ細かさを実現するのは困難である。さらに、我々は target ポイントカット記述子を用いて対象となるメンバアクセスのあった構造体の情報を利用することで、不必要な ARP パケットについてタイムスタンプを記録せずにすんでいる。

4.2.2 節のケーススタディは コンテキストスイッチが行われると、プロセスの情報を入れている task_struct 構造体の timestamp メンバの値を現在の時刻に更新されることを利用している。task_struct 構造体の timestamp メンバへのアクセスをポイントカットすることで、プロセスが切り替わる瞬間の時刻を調査している。同じことを execution ポイントカット記述子のみで調査することも一応可能である。しかし、プロセスの切り替えを実際に行っている sched_info_switch 関数は static inline 関数なので選択できない。よって、この関数を含む schedule 関数をポイントカットすることになるが、schedule 関数は大きな関数なので時間計測の精度がその分下がるという問題がある。

5. 関 連 研 究

5.1 C/C++用の動的アスペクト指向システム

C 言語用の動的アスペクト指向システムはいくつかあるが、そのいずれも構造体のメンバアクセスをポイントカットすることができない。構造体メンバへのアクセスをポイントカットできることは OS カーネルのプロファイリングやデバッグをするために必要な機能である。

TOSKANA[9] は NetBSD 用の動的アスペクト指向システムである。KLASY 同様、アスペクトをウィーブすることで実行中のカーネルを動的に書き換える事ができる。しかし、TOSKANA では普通の C コンパイラの作成したシンボル情報のみを用いているため、メンバアクセスをポイントカットすることはできない。

TOSKANA-VM[10] も実行時に動的にアスペクトをカーネルにウィーブするシステムである。TOSKANA-VM の手法は、動的にウィーブできるようにするために改造された Java 仮想機械（VM）である Steamloom[3),11)] の手法と似ている。TOSKANA-VM のカーネルは特殊なコンパイラを使って LLVM という VM 上で動く機械語にコンパイルされる。この機械語には詳しいシンボル情報があるため、TOSKANA-VM では変数の読み書きのような様々なジョインポイントをポイントカットすることができる。しかし、カーネルが VM の上で動作するため、このアプローチは実際のハードウェアで動いているカーネルをプロファイリングする用途には向かない。

DAC++[1] と TinyC2[2,4] は C++で書かれたユーザプログラムに対する動的アスペクト指向システムである。アスペクトをウィーブするため、これらのシステムは実行中の C++プログラムを書き換える。通常のコンパイラで生成されたシンボル情報をこれらのシステムは用いているため、これらのシステムでは関数の呼び出し箇所をジョインポイントとするポイントカットしか提供していない。

Arachne[8] は C 言語で書かれたユーザプログラムに対する動的アスペクト指向システムである。Arachne は通常のコンパイラでコンパイルされたシンボル情報だけを用いているが、グローバル変数や malloc で確保されたメモリブロックに対するポイントカット記述子を提供している。しかし、Arachne では構造体のメンバアクセスのポイントカットは提供していない。また、ページフォルトを用いてメモリブロックへのアクセスを検出しているため、メモリブロックへのアクセ

スをポイントカットすると大幅に性能劣化する。

5.2 C/C++用の静的アスペクト指向システム

ソースレベルの情報のほとんどの部分はコンパイル時に失われるため、既存の動的アスペクト指向システムでは非常に少ないシンボル情報しかウィーブ時に利用できない。一方、静的アスペクト指向システムではウィーブするときにソースレベルの情報を完全に利用できる。そのため、様々なポイントカット記述子を提供できると考えられる。しかし、2 章で述べたように静的アスペクト指向システムは OS カーネルのプロファイリングやデバッグには不向きである。なぜなら、カーネル開発者は再起動することなく新しいアスペクトをウィーブする必要があるからである。

AspectC は初期の C 言語用の静的アスペクト指向言語であり、これを用いた研究[5)~7)] により、アスペクト指向プログラミングが OS カーネルをモジュール化するのに役立つことが示されている。例えば、ディスクブロックを先読みするプログラムは仮想記憶サブシステムとディスクサブシステムの両方にまたがったコードとなる。先読みのプログラムをサブシステムと切り離して書くにはアスペクト指向プログラミングが必要である。

5.3 その他の関連ツール

動作中の OS カーネルを書き換えるツールがいくつかある。KLASY のバックエンドに利用している Kerninst[20)] もそのようなツールの一つである。しかし、Kerninst の提供する抽象度はソースレベルではなく、アセンブリレベルであり、その利用者は関数内の調査したい箇所やローカル変数のメモリアドレスを自分で調べなくてはならない。ソースレベルの抽象度で調査できた方が効率がよいので、これは OS カーネルのデバッグやプロファイリングを行うには重大な欠点となる。Kerninst と同様のツールとして GILK[17)] というものがあり、フックとしてジャンプ命令だけを用いる。GILK の方が Kerninst よりも性能が良いが、GILK は古い Linux カーネルにしか対応していない。

LKST[13)]、DTrace[4)]、SystemTAP[18)]、LTT[22)] はすべてカーネル内のイベント発生箇所でログを取得するシステムである。これらのシステムの利用者はログを取得するかどうか動的に制御することができる。しかし、ログを取得できる点は OS カーネルコンパイル時に静的に決まっており、利用者はそれらの決まった箇所からログを取得する点を選ぶことしかできない。

6. ま と め

本論文では動的アスペクト指向システムである KLASY を提案した。KLASY は *souce-based binary-level dynamic weaving* という手法を用いて、構造体のメンバへのアクセスをポイントカットすることができる。プロファイリングしたい箇所に関連する多くの関数を選ぶよりも関連する構造体をいくつか選ぶ方が簡単であるため、構造体のメンバへのアクセスをポイントカットできることは重要である。 KLASY はローカル変数やポイントカットされたメンバを持つ構造体を参照するためのポイントカット記述子も提供している。KLASY は主にプロファイリングやデバッグに用いられることを想定しているため、これら実行時コンテキストを参照できることは必要である。ポイントカットで選択されたジョインポイント及び対象の構造体のメモリアドレスをウィーブ時に見つけられるようにするため、我々は C コンパイラを改造して、拡張したシンボル情報を出力するようにした。ウィーブ時には実行中のカーネルを動的に書き換えて調べたアドレスの箇所にフックを埋め込み、アドバイス本体を呼び出す。

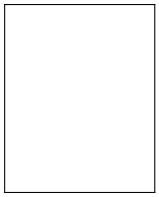
本論文ではこの手法の限界についても分析した。本手法の問題の一つは、拡張したシンボル情報を出力しなければならないため、改造した gcc が元の gcc ほど最適化を実行できないことである。これによる性能劣化は平均 1%である。アスペクトをウィーブした際のオーバーヘッドについても測定したが、ブレークポイントトラップによるフックがベンチマーク開始から終了までに 9 千万回呼ばれるなど過度にある場合は大幅に性能が劣化することがわかった。また、本手法のもう一つの問題は、コンパイラが最適化のためにコードを移動させたり消したりした場合に、ポイントカットで選択されたジョインポイントのアドレスが見つけらず、それらの箇所にフックを埋め込めないことである。本研究では x86 アーキテクチャ上の Linux に対して実装を行ったが、本研究の手法は他の OS やアーキテクチャでも実現可能だと考えられる。これらの改善、検討は今後の課題である。

参 考 文 献

1) Almajali, S. and Elrad, T.: Coupling Availability and Efficiency for Aspect Oriented Runtime Weaving Systems, *Proc. of DAW05*, AOSD 05 (2005).
2) Apache Software Foundation: Apache Tomcat, `http://tomcat.apache.org/`.
3) Bockisch, C., Haupt, M., Mezini, M. and Ostermann, K.: Virtual machine support for dynamic join points, *Proc. of AOSD '04*, New

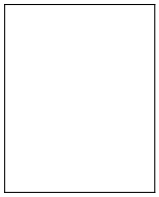
York, NY, USA, ACM Press, pp.83–92 (2004).
4) Cantrill, B.M., Shapiro, M.W. and Leventhal, A.H.: Dynamic Instrumentation of Production Systems, *Proc. of the USENIX Annual Technical Conference*, USENIX Association, pp.15–28 (2004).
5) Coady, Y. and Kiczales, G.: Back to the future: a retroactive study of aspect evolution in operating system code, *Proc. of AOSD '03*, ACM Press, pp.50–59 (2003).
6) Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N. and Ong, J. S.: Structuring operating system aspects: using AOP to improve OS structure modularity, *Commun. ACM*, Vol.44, No.10, pp.79–82 (2001).
7) Coady, Y., Kiczales, G., Feeley, M. and Smolyn, G.: Using aspectC to improve the modularity of path-specific customization in operating system code, *Proc. of ESEC/FSE-9*, ACM Press, pp.88–98 (2001).
8) Douence, R., Fritz, T., Loriant, N., Menaud, J.-M., Ségura-Devillechaise, M. and Südholt, M.: An expressive aspect language for system applications with Arachne, *Proc. of AOSD '05*, New York, NY, USA, ACM Press, pp. 27–38 (2005).
9) Engel, M. and Freisleben, B.: Supporting autonomic computing functionality via dynamic operating system kernel aspects, *Proc. of AOSD '05*, New York, NY, USA, ACM Press, pp.51–62 (2005).
10) Engel, M. and Freisleben, B.: Using a Low-Level Virtual Machine to Improve Dynamic Aspect Support in Operating System Kernels, *Proc. of ACPIS*, AOSD (2005).
11) Haupt, M., Mezini, M., Bockisch, C., Dinkelaker, T., Eichberg, M. and Krebs, M.: An execution layer for aspect-oriented programming languages, *Proc. of VEE '05*, New York, NY, USA, ACM Press, pp.142–152 (2005).
12) Hibino, H., Kourai, K. and Chiba, S.: Difference of Degradation Schemes among Operating Systems, *Proc. of Workshop on Dependable Software - Tools and Methods*, DSN-2005, pp. 172 – 179 (2005).
13) Hitachi, Ltd. and Fujitsu, Ltd.: Linux Kernel State Tracer (2001, 2005). `http://lkst.sourceforge.net/`.
14) Lehey, G.: Improving the FreeBSD SMP implementation, *Proc. of the FREENIX Track: 2001 USENIX Annual Technical Conference*, USENIX Association, pp.155–164 (2001).
15) Masuhara, H., Kiczales, G. and Dutchyn, C.: Compilation Semantics of Aspect-Oriented Programs, *Proc. of FOAL Workshop*, AOSD 2002, pp.17–26 (2002).
16) Molloy, S. and Honeyman, P.: Scalable Linux Scheduling, *Proc. of the FREENIX Track: 2001 USENIX Annual Technical Conference*, USENIX Association, pp.285–296 (2001).
17) Pearce, D. J., Kelly, P. H. J., Field, T. and Harder, U.: GILK: A Dynamic Instrumentation Tool for the Linux Kernel, *Computer Performance Evaluation / TOOLS*, pp. 220–226 (2002).
18) Prasad, V., Cohen, W., Eigler, F. C., Hunt, M., Keniston, J. and Chen, B.: Locating system problems using dynamic instrumentation, *Proc. of the Linux Symposium*, Vol.2, pp.49–64 (2005).
19) Roberson, J.: ULE: A Modern Scheduler for FreeBSD, *Proc. of BSDCon '03*, USENIX Association, pp.17–28 (2003).
20) Tamches, A. and Miller, B. P.: Fine-grained dynamic instrumentation of commodity operating system kernels, *Proc. of OSDI '99*, Berkeley, CA, USA, USENIX Association, pp.117–130 (1999).
21) Tux.Org, Inc: UnixBench. `http://www.tux.org/pub/tux/niemi/unixbench/`.
22) Yaghmour, K. and Dagenais, M.R.: Measuring and Characterizing System Behavior Using Kernel-Level Event Logging, *Proc. of the USENIX Annual Technical Conference*, pp.13–26 (2000).
23) Yamamura, S., Hirai, A., Sato, M., Yamamoto, M., Naruse, A., and Kumon, K.: Speeding Up Kernel Scheduler by Reducing Cache Misses - Effects of cache coloring for a task structure -, *Proc. of the FREENIX Track: 2002 USENIX Annual Technical Conference*, USENIX Association, pp.275–286 (2002).
24) Zhang, C.: TinyC^2: Towards Building a Dynamic Weaving Aspect Language for C, *Proc. of FOAL 2003*, AOSD 2003 (2003).

(平成 18 年 12 月 15 日受付)
(平成 19 年 4 月 1 日採録)



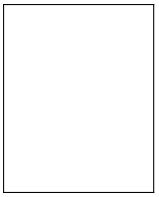
柳澤 佳里（学生会員）

1980 年生．2003 年東京工業大学理学部情報科学科卒業．2005 年東京工業大学大学院情報理工学研究科数理・計算科学専攻修士課程終了．オペレーティングシステム，計算機ネットワーク，言語処理系など基盤システムの研究に従事．



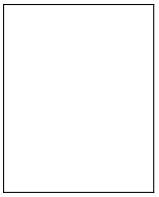
光来 健一（正会員）

1975 年生．2002 年東京大学大学院理学系研究科情報科学専攻博士課程修了．同年日本電信電話株式会社入社，未来ねっと研究所勤務．2003 年より東京工業大学大学院情報理工学研究科数理・計算科学専攻助手．博士（理学）．オペレーティングシステム，ネットワークの研究に従事．日本ソフトウェア科学会，ACM 各会員．



千葉 滋（正会員）

1968 年生．1991 年東京大学理学部情報科学科卒業．1996 年東京大学大学院理学系研究科情報科学専攻博士課程退学．東京大学助手，筑波大学講師を経て，現在東京工業大学大学院情報理工学研究科助教授．博士（理学）．言語処理系およびオペレーティングシステム等システムソフトウェアの研究に従事．日本ソフトウェア科学会，ACM 各会員．



石川 零

1981 年生．2004 年東京工業大学理学部情報科学科卒業．2006 年同大学院情報理工学研究科数理・計算科学専攻修士課程修了．現在キヤノン株式会社に勤務．