Vol. 48 No. SIG 0(PRO 35) 情報処理学会論文誌：プログラミング Apr. 2007

# XenLASY: Xen のI/O処理を追跡するための アスペクト指向プロファイラ

柳澤佳里[†] 光来健一[†] 千葉滋[†]

本論文では Xen における仮想マシン（ドメイン）にまたがった I/O 処理をプロファイリングするための動的アスペクト指向システム XenLASY を提案する。Xen 上ではドメイン 0 とドメイン U と呼ばれる 2 種類の仮想マシンが動作しており、ドメイン U の OS が I/O 処理を行う際にはドメイン 0 の OS を介して行われる。この際に、制御フローが途切れ、データ形式も変わるため、データフローをたどったプロファイリングが困難である。 XenLASY は xflow ポイントカットという Xen 上のデータフローを選択するポイントカットを提供しており、ドメイン 0 とドメイン U にまたがるデータフローでもこれを用いることでドメインを越えた I/O フローを追跡可能である。また、xflow ポイントカットを用いるとデータ構造が変わった場合でも追跡が可能である。この実現のために XenLASY はアスペクトにより各ドメインに同時にプロファイリングコードを織り込む。xflow ポイントカットでは、処理するデータの始点、中継点、終点を指定することが出来るため、調査したいところだけに絞って調査することが可能である。また、この指定には名前をつけることが出来、再利用が可能である。実験の結果、XenLASY のデータフロー追跡機能が現実的な時間で動くことがわかった。また、ネットワーク I/O を追跡するケーススタディにて XenLASY を用いることで時間がかかっている箇所を発見することができた。

# XenLASY: Aspect-oriented Profiler for Tracing I/O Processing on Xen

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

In this paper, we propose an aspect-oriented system for profiling I/O process between domains on Xen. There are two kind of virtual machines working on Xen. I/O process of domain U is always passed through domain 0. Since control flows are interrupted and data structures change through I/O process, developers are hard to find the bottlenecks by tracing I/O process. XenLASY enables developers to trace I/O process between domains by using xflow pointcut. It enables developers to record data flow even if data structures are changed. XenLASY inserts profiling code into each domain to realize this feature. Since the developers can instruct points to start, transit and quit tracing, they can ignore insignificant data. Since the instruction can be named, they can reuse it. We implemented XenLASY on Xen. According to the experiment, we found that our faculties for tracing I/O processing runs in acceptable time. Our case study shows that XenLASY helps investigating I/O bottlenecks in Xen.

## 1. はじめに

近年、消費電力やスペースの削減のために仮想マシンが使われるようになってきている。実際、近年の Intel や AMD から発表されるプロセッサには仮想化を支援する仕組みが搭載されており、VMWare Server や Virtual PC など無償で簡単に仮想マシンを作れるようになってきている。仮想マシンを使うと従来では複数台必要だった計算機が 1 台ですむようになる反面、従来の OS のチューニングがうまく働かなくなる危険性がある。実際、ディスク I/O 性能を向上する技術である anticipatory scheduling[9] が VM ではうまく働かないという事例が報告されている。[10] そのため、システム全体で I/O 処理をチューニングする必要がある。

特に Xen[3] の場合は I/O を行うときにドメイン 0 を通るため、ボトルネックとなる箇所が増加する。ドメイン 0 は物理デバイスを制御する仮想マシン (VM) であり、Xen の他の VM (ドメイン U) からの通信はすべてドメイン 0 を通して行われる。ここで、ドメイン U からドメイン 0 に処理を依頼する部分、ドメイ

† 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, Tokyo Institute of Technology

ン 0 でのスケジューリングのタイミングなどオペレーティングシステム (OS) 単体では無かった箇所がボトルネックとなりうる。

このようなボトルネックを発見するツールとして XenLASY を提案する。XenLASY は我々が開発しているカーネル用アスペクト指向システム KLASY[16),17)] を拡張して実装したプロファイラである。KLASY が持つポイントカットに加えて、xflow ポイントカットという Xen 上で行われる I/O 処理のデータフローを選択するポイントカットを持つのが特徴である。これにより、Xen 上のデータフローの追跡をアスペクトとして簡単に書ける。xflow ポイントカットを用いると OS カーネル内で調査していたデータが他のドメインに渡された場合や複製された場合でも調査を継続することが出来るようになる。XenLASY では各ドメインの OS カーネルにアスペクトを織り込み、そのプロファイリングが可能である。

我々は XenLASY を用いてネットワーク I/O を追跡するケーススタディーを行った。その結果、ドメイン 0 のネットワーク処理において長い時間がかかっていることを発見した。また、ケーススタディを通し、データフローを追跡する機能があることで KLASY で単純に調査する場合よりメモリー使用量が大幅に減らせることを確認した。

以下、2 章にて VM を考慮したカーネルのチューニングの重要性と難しさを論じ、3 章にて Xen 上で処理の流れを追跡するためのアスペクト指向システムである XenLASY を提案する。4 章では XenLASY の実装について述べ、5 章では XenLASY 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。6 章では関連研究について論じ、7 章で本論文をまとめる。

## 2. VM を考慮したカーネルのチューニング

従来、オペレーティングシステム (OS) にて I/O 処理のチューニングをすることで、性能を向上させることができた。なぜなら、全ての物理デバイスは OS の管理下にあり、OS 上で動作するプロセスの挙動も完全に把握することが出来たからである。もし実行が遅い場合には、プロファイリングによりボトルネックを発見して改良することで、ボトルネックを解消する。従来の性能向上の例として、ディスクのシークを減らすために、リード要求を処理した後、すぐに同じプロセスが近隣セクタへのリード要求を出すかも知れないと予測して暫く待つ anticipatory scheduling[9)] がある。

仮想マシン (VM) を使う場合、OS 単体でなくシステム全体で I/O 処理をチューニングする必要が出

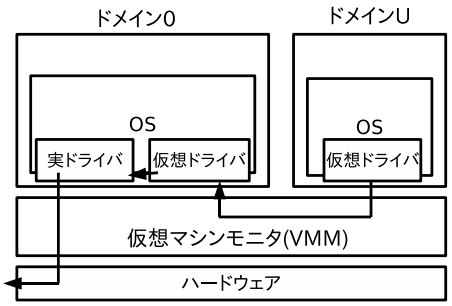



**図 1** Xen の I/O 処理の流れ

てくる。一つの物理デバイスが複数の VM で共有されるからである。この場合、一つの VM でチューニングをして最適な状態を作ったとしても全体として最適な状態になるとは限らない。例えば、anticipatory scheduling は VM 上ではうまく動かない。[10)] Anticipatory scheduling を行うにはディスクのヘッドの位置を見積もる必要があるが、VM からは他の VM 上で行われる I/O がわからないため、ヘッドの位置の見積もりを正しくできずシークが頻発するからである。

特に Xen[3)] の場合はドメイン 0 を考慮したチューニングが必要となる。ドメイン 0 とは、Xen で物理デバイスを操作できる特権 VM である。ドメイン 0 以外の VM は全てドメイン U と呼ばれ、図 1 のように、これらドメイン U で I/O 処理を行う場合は全てドメイン 0 を介して行われる。このように、ドメイン U の I/O 処理が全てドメイン 0 を通るので、OS 単体の場合に比べてボトルネックとなりうる箇所が増加する。例えば、ドメイン U からドメイン 0 に処理を依頼する箇所、ドメイン 0 がスケジューリングされるタイミング、ドメイン 0 でのデバイスアクセスなどが新たにボトルネックとなりうる。実際、ドメイン U からドメイン 0 に処理を依頼する箇所がボトルネックとなった事例が報告されている。[13)] また、複数のドメイン U が I/O 処理を行う場合には、これらドメイン U 間での競合が起こりえる。この場合、ドメイン 0 でデバイスアクセスの順番待ちが起きることになる。

このような状況でチューニングを行うにはドメイン U とドメイン 0 にまたがって I/O 処理の流れ (フロー) を調べられる必要がある。なぜなら、ボトルネックとなりうる箇所はシステム全体に渡っており、個々の I/O 処理について原因を調べる必要があるからである。個々の I/O 処理を識別できるれば、調べる必要のある I/O フローだけを調査することが出来る。しかし、関数呼び出しの流れを追うコールフローだけでこれを行うのは不十分である。なぜなら、ドメイン間の

通信はコールフローでは追えないからである。また、カーネル内の処理はデバイス関連の処理をするボトムハーフとユーザーランドに近い所で処理をするトップハーフに分かれているが、ここでもコールフローを追うことが出来ない。ボトムハーフはデータをキューにいれて処理を終了し、別スレッドで動くトップハーフがキューから取り出して処理を引き継ぐようになっているからである。

一方、データのポインタを追跡する単純なデータフローだけでも処理の流れを追跡するのには不十分である。まず、この方法ではデータの形式が変わると追えなくなる。ドメイン間の通信を行う際には Linux のネットワークバッファである sk_buff 構造体から共有メモリー上のバッファの生データに変換されるため、この方法ではドメインを越えた追跡が出来ない。また、データの複製や分割が起きた場合もこの方法では追えなくなる。例えば、TCP の再送用にデータが複製されることや MTU のサイズにあうようにパケットが分割されることがあり、この場合は複製や分割されたデータを追跡することが出来なくなる。

## 3. XenLASY

このような問題を解決するため我々は Xen 上で行われる I/O 処理を追跡するためのアスペクト指向プロファイラ XenLASY を提案する。XenLASY はドメインを越えたデータフローの追跡を行うために xflow ポイントカットを提供している。アスペクトの織り込みに際して、XenLASY は各ドメインの OS にのみコードを埋め込み、仮想マシンモニタ (VMM) には埋め込まない。本システムは I/O データのフロー追跡に特化したプロファイラであるため、この制限は大きな問題にはならないと考えられる。なぜなら、処理を依頼する際に VMM が行うことは依頼されたドメインにイベントを送る程度の単純な処理であり、依頼元ドメインが依頼してから依頼先ドメインがイベントを受け取るまでの間にボトルネックがあるか否かは、前後の処理を行うドメイン上の OS でログを取ることで、発見が可能だからである。

### 3.1 KLASY

我々は以前開発した KLASY[16),17)] を拡張して XenLASY を開発した。KLASY は OS カーネル用の動的アスペクト指向システムであり、その特徴は source-based binary-level dynamic weaving が出来ることである。これは、ソースコード上の情報を元に実行時にアスペクトを織り込む機能である。C 言語では型の情報がコンパイル時に消えてしまうが、KLASY ではコンパイル時にそれを拡張したシンボル情報に記録し、実行時にはこのシンボル情報を元にアスペクトを織り込む。特に構造体のメンバーにアクセスした箇所を調査点として選択 (ポイントカット) できる。このポイントカットのことを access ポイントカットという。ポイントカットされた箇所の前や後でアドバイスを実行させてプロファイル情報を記録することができる。OS カーネルの実行時に動的にアスペクトを織り込むことで、調査を行う必要があるときにすぐに調査が行え、再コンパイル、再起動の後で現象が再現するのを待つ必要が無くなる。また、アスペクト指向を用いることで調査のコードを 1 つのモジュールとしてまとめて記述することが出来る。

### 3.2 xflow ポイントカット

xflow ポイントカットとは Xen 上の I/O 処理のデータフローを選択するために我々が新たに提案するポイントカットである。xflow ポイントカットは Xen 上ドメインのある実行点で生成されたデータが、それを格納するデータ形式の変遷をたどって消滅するまでの流れに、指定したデータが入っている時を選択するポイントカットである。データの変遷はデータにフロー ID という識別子を割り振って管理し、データの形式が変わった場合もそれが引き継がれる。

アスペクト指向言語の設計では、適切な抽象度のポイントカットを提供することが重要である。xflow に似たポイントカットとして AspectJ は cflow を提供しているが、これは指定した関数が呼び出されて、それが終了するまでを追跡するのに使われる。一方、xflow はデータの流れを追跡し、データを処理する関数を追うのに使われる。xflow はデータの流れを追うため、それを処理するスレッドが変わっても追跡できる。さらに、データの形式や、データを処理している OS があるドメインが変わった場合でも明示することで追跡を継続することが出来る。xflow が利用できなくても、適当なライブラリを作り、煩雑であるが手書きで同等のアスペクトを書くことは cflow 同様可能である (なお、後述するように、手書き部分を単純なマクロで記述することは出来ない)。しかし、追跡したいフローを xflow で一度定義すれば、xflow を再利用して望みのアスペクトを簡潔に書くことが出来る。

図 2 は xflow ポイントカットを用いたアスペクトの例である。ポイントカット記述は advice ブロックの中にある pointcut で囲まれたブロックに記述し、その点で実行するコードは before や after で囲まれたブロックに記述する。この例ではデータフローの種類としてネットワーク I/O を行うデータのフローである

```
<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>
```

**図 2** xflow を用いたアスペクトの例

netflow という名前のフローを選択している。そして、access ポイントカットにより sk_buff 構造体の任意のメンバ (%) をアクセスしている時点で、その sk_buff 構造体インスタンスが netflow データフローのデータならアドバイスが実行される。

このように、データのフローをアスペクトで選択するのには xflow ポイントカットにフローの種類を示す名前と変数名を与えることで行う。この例では sk_buff 構造体が入っている変数が target ポイントカットで skb として取り出され、それが netflow という名前で定義されたフローに含まれていた場合に選択される。netflow という種類のフロー中の各々のフローを識別する ID は xflow ポイントカットの第 3 引数にて指定した変数で取り出すことができる。これを用いて netflow と名付けられた種類のフローの各々のデータフローを識別することが出来る。なお、フロー識別子を使わない場合は xflow ポイントカットの第 3 引数を省略できる。

この例のアドバイスでは、その場所のプログラムカウンタ ($pc$) とフロー識別子 (id)、時刻 (tsc) を記録する。なお、図のコードで出てくる DO_RDTSC は現在のタイムスタンプカウンタの値を取得するマクロであり、STORE_DATA3 は後でユーザーランドからデータを取り出せるようデータを保存するマクロである。STORE_DATA3 には保存するデータの数により STORE_DATA1(1 つの場合) や STORE_DATA2(2 つの場合) などのバリエーションがある。このようにすることで、各フローのデータがいつどこを通ったかを後で調べられるようになる。

### 3.3 xflow ポイントカットの定義

図 3 は xflow ポイントカットの定義の例である。この例は alloc_skb_from_cache 関数で生成された sk_buff 構造体のデータの流れが skb_clone で複製され__kfree_skb で終了するまでを追跡している。図のような記述を advice ブロックとは別に書き、xflow を定義する。xflow の名前は name 属性で定義し、advice にて xflow を選択するのに使う。

xflow の定義ではデータを追跡しはじめる始点 (start) や追跡をやめる終点 (quit) を指定する機構がある。また、データの形式が変わった場合のために、変化後に追跡を継続するための中継点 (transit) を指定できる。これによりスレッドやドメイン、データ構造が変わっても追跡でき、手動で指定できることで無駄な追跡をしないという利点がある。始点や終点ではポイントカットなどの記述により、どの実行点でどのデータを追跡、追跡終了するかを指定する。このために始点ではポイントカットで選択した構造体インスタンスの先頭へのポインタを鍵として新たに生成したフロー ID を引けるようデータベースに登録する。終点では構造体インスタンスの先頭アドレスからフロー ID を得る対応付けを削除するため、データベースからエントリを削除する。

xflow の定義では複製やデータ形式の変更後もフロー ID を取得できるようにするため、中継点 (transit) を記述できる。例えば、TCP の再送を実現するために sk_buff 構造体インスタンスが複製されるが、単純にデータを追うだけではこの複製を追跡することができない。そこで、transit を記述することで複製やデータ形式の変更が行われる点で元となる構造体インスタンスのフロー ID を複製先あるいは変換先の構造体インスタンスのフロー ID としてデータベースに登録する。

これらの点の選択はポイントカットで行う。start および quit ではデータベースに登録あるいはデータベースから削除する構造体インスンタンスの格納場所を示す記述を省略でき、省略した場合には access ポイントカットで選択した構造体のインスタンスが選ばれる。これ以外を選択するには select 要素を追加し、その属性 local_var にて構造体インスタンスの先頭アドレスが入った変数を指定する。例えば、

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

と書くと、変数 data が指している構造体インスタンスがフロー ID を引くための鍵となる。なお、用意されている方法でフロー ID を登録、削除するのでは不足する場合に備え、action という任意のコードを記述する方法を用意している。この方法は全てのプログラムを自分で書く必要があるため推奨されない。

中継点をあらわす transit 要素では中継点でどの構造体インスタンスに対応していたフロー ID をどの構造体インスタンスに対応づけるかを move 要素で明示する。この例では、from 属性に skb、to 属性に n が書いてあるので、変数 skb に対応するフロー ID が選

```
<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>
```

**図 3** xflow 定義の例

択されたポイントカットで変数 n に引き継がれる。そして、変数 skb からはフロー ID が引けなくなる。もし、変数 skb と変数 n の両方にフロー ID を引き継ぐ場合は move ではなく copy を用いる。copy で記述すると変数 skb に対応するフロー ID を変数 skb と変数 n の両方が引けるようになる。

### 3.4 ドメインを跨ぐ xflow ポイントカットの定義

Xen のドメインをまたがった transit を行えるようにするため、我々は transit 要素にて中継方法を指示する xin_move 要素、xin_copy 要素、xout_move 要素、xout_copy 要素を用意した。3.3 節では中継点 (transit) の記述方法を示したが、この記述ではドメインをまたいでフロー ID を伝搬させることが出来ない。なぜなら、構造体インスタンスの先頭アドレスからフロー ID を得ているため、アドレス空間が違い、データベースを共有していない他のドメインではアドレスからフロー ID を引くことが出来なくなるからである。また、ドメインを跨ぐ場合は通信バッファに入る時点でデータ構造が大きく変わるため単純に構造体インスタンスにフロー ID を結びつける方法ではうまくいかない。

図 4 はドメインを跨ぐ際の xflow 定義の例である。これはネットワーク I/O のデータがドメイン U からドメイン 0 に渡されるときにフローを追跡できるようフロー ID をドメイン U からドメイン 0 に伝搬させるための記述である。Xen でネットワーク I/O やディスク I/O を行う場合はドメイン内で使われてきたデータがヘッダ部分とデータ部分に仮想デバイスドライバで分割される。そして、データ部分は共有メモリ経由で宛先のドメインに渡される。データ部分が共有メモリのどこに入っているかは別途渡されるヘッダ部分にて知らせるようになっている。そこで、ドメインを跨ぐ場合はヘッダにフロー ID を格納し、送り先でそれを取り出すことでフロー ID を伝搬させる。この例ではドメイン U のホストである linuxU の netfront.c ファイルで netif_tx_request 構造体の flags メンバーにアクセスするところでネットワーク I/O に用いられるヘッダである変数 tx にフロー ID を格納している。そして、ドメイン 0 の netback.c ファイル中で netif_tx_request 構造体の flags メンバーにアクセスするところで tx に格納されていたフロー ID を変数 skb に対応づけている。これは xin_move や xout_move で指示している。

xin_move 要素ではどのようにフロー ID をヘッダーに格納し、xout_move 要素ではどのように取り出すかを指示する。xin_move 要素の name 属性は後で格納方法を参照するための名前を指定する。この名前は xout_move 要素の name 属性で指定される。xin_move 要素中の from 属性はどの変数のフローのフロー ID を用いるかを示し、to 属性はどの変数がフロー ID を格納するヘッダかを示している。xin_move に続く field 要素はヘッダ中のどのメンバーの何ビット目からどのサイズまでをフロー ID を格納するのに使うかを示している。図 4 では、

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

とあるので、flags メンバーの下位 4 ビット目から 12 ビットを用いることが指示されている。なお、指示されたフィールドだけで収まりきらなかった場合は上位のビットが不足分だけ失われることになる。時刻に基づいて並べるとデータが実行点を通過した時刻がわかるため、フロー ID のうち上位ビットが多少失われてもデータの流れを追うのには支障がないと考えられる。xin_move では格納した時点で skb に割り当てられていたフロー ID は消去されるが、xin_copy を用いるとこれを残しておくことが出来る。xout_move に対しても同様に xout_copy が用意されている。この機能により、処理するスレッドが変わった場合や、データ形式が変わった場合だけでなく、データを扱う OS のあるドメインが変わった場合でもデータの流れを追跡できる。

図 4 の例では選択するジョインポイントがどのドメインにあるかを指定するために@を選択されるジョインポイントを特定のファイルに制限する within_file ポイントカットで記述している。例えば、linuxU というホスト名のドメインの netfront.c ファイルを選択する

```
<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="tx" to="skb" />
</transit>
```

**図 4** ドメインをまたがる xflow 定義

には netfront.c@linuxU と記述する。なお、@によるホストの選択は選択するジョインポイントを特定の関数に制限する within_function ポイントカットの中でも使うことが出来る。@が無い場合は全てのドメインが選択される。

## 4. KLASY の Xen への対応

我々はカーネル用アスペクト指向システムである KLASY を改造して、XenLASY を実装した。改造の要点は、アスペクト言語を拡張し xflow を使えるようにすることと、アスペクトの分配方法の提供、KLASY 内部で使われている実行中の OS カーネルを操作するプログラムである Kerninst[15] を Xen 上ドメイン内の OS で動くようにすることの 3 点である。本節ではそれら改造点について述べる。

### 4.1 アスペクト言語の拡張

本節では、まず 3 節で示した xflow 定義からどのようなコードが実際に織り込まれるかを示す。次に、xflow ポイントカットがどのように変換されるかを示す。なお、xflow 定義から作られたコードは一般のアスペクトに先んじて織り込まれる。これにより、アスペクトで xflow ポイントカットを用いた際には織り込みの順序を気にすること無くフローを選択できるようになる。

例えば、図 3 の start は図 5 のようなコードになる。これは図 3 の start にあったポイントカット記述に target ポイントカットを加え、アドバイスとして target ポイントカットで選択したポインタを新たに生成したフロー ID のフローとして登録している。ここでは get_new_flowid 関数により、新しいフロー ID を得て、それから register_flowid 関数によって構造体インスタンスの先頭アドレスを鍵としてフロー ID を引けるようにデータベースに登録する。もし、select があった場合は target ポイントカットの代わりに local_var ポイントカットが使われる。この場合はローカル変数そのものを登録に用いるポインタとして扱う。

```
<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>
```

**図 5** start のコード

```
<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>
```

**図 6** transit のコード

図 3 の transit は図 6 のようなコードになる。これは図 3 の transit にあったポイントカット記述に local_var(skb, random_str1) AND local_var(skb, random_str2) が加わったポイントカットになることで、ローカル変数への参照を取得し、move 要素の from 属性である skb に割り当てられていたフロー ID を to 属性である n に割り当てるようにしている。ここで、get_flowid 関数を用いて構造体インスタンスの先頭アドレスを鍵としてフロー ID を取得している。なお、move を用いているので n に割り当てた後は skb からはフロー ID が引けなくなる。copy の場合は最後の remove_flowid(skb) をせず、フロー ID が引けるままにしておく。

図 4 の 1 つ目の transit は、図 7 のようなコードに

```
<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 = random_str2;
  int id = get_flowid(netflow, skb);
  tx-&gt;flags |= id &lt;&lt; 4;
  id &gt;&gt;= 12;
  remove_flowid(netflow, skb);
 </before>
</aspect>
```

**図 7** ドメインをまたがる xflow のコード

なる。xin_move の場合も from 属性や to 属性で与えられた変数への参照を取得する。xin_move はヘッダにフロー ID を格納するためにまず get_flowid 関数で skb に割り当てられているフロー ID を取得し、field 要素で指示された箇所にフロー ID を埋め込む。なお、フロー ID を取得した後は取得元から ID が得られないよう remove_flowid 関数を用いて初期化している。remove_flowid 関数は構造体インスタンスの先頭アドレスを鍵としてフロー ID とのマッピングを削除する。2 つ目の transit にある xout_move では xin_move とは逆の動きを行い、変数 skb の指す構造体インスタンスを鍵にして埋め込まれたフロー ID が引けるようデータベースに登録する。quit は start と同様に変換され、register_flowid 関数の代わりに remove_flowid 関数を呼んでデータベースからマッピングを削除する。

xflow ポイントカットを用いた図 2 のようなアスペクトは図 8 のようなコードになる。xflow ポイントカットが織り込まれる際は get_flowid 関数を用いたコードになる。そして、get_flowid 関数でフロー ID が得られなかった場合はアドバイスを実行しない。ここで、xflow ポイントカットの第 3 引数が省略された場合はフロー ID を入れる変数名としてプログラム内でぶつからないランダムな文字列が用いられる。

xflow ポイントカット定義や xflow ポイントカットを単純なマクロ変換で行うのは難しい。なぜなら、xflow ポイントカット定義では抽象度を保つために name 属性にて xflow に名前をつけ、実際にポイントカットで用いたときにそれが正当なものか判定するからである。このような仕組みは xin_move、xin_copy で指定した格納方法を xout_move、xout_copy にて参照するのにも用いられている。また、xflow ポイントカットを

```
<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>
```

**図 8** xflow ポイントカットのコード

実装するにも変数が使えることををコンパイル時に確認するためにポイントカット全体を構文解析する必要があるため、マクロ変換で先に述べたようなアドバイスコードを作るのは難しい。さらに、xflow ポイントカット定義が略記を許す文法になっていることも状態を持つ必要性を生み、マクロ変換での実装を難しくしている。

### 4.2 アスペクトの分配

XenLASY ではアスペクトのコードが自動的に各ドメインに分配されるため、アスペクトの集中管理が行える。XenLASY におけるアスペクトの各ドメインへの織り込みはポイントカットで指示された通りに自動的に行われる。各ドメインではアスペクトを織り込むためのランタイムが動いており、アスペクトを織り込む際にはコンパイル後にそれらランタイムにコンパイルされたアドバイスとポイントカットから作られたフック挿入箇所一覧が送られる。それから、ランタイムが各ドメインのカーネルにアドバイスをロードし、その後フック挿入箇所一覧に基づきフックを挿入する。within_file ポイントカットや within_function ポイントカットで@が使われた場合は、これを含むポイントカットは指定されたドメインのフック挿入箇所一覧のみに反映される。

### 4.3 Kerninst の改造

Kerninst を Xen 上ドメイン内の OS に持っていってもそのままでは動かなかったため、次の 2 点を変更した。一つは Kerninst のカーネルモジュールから特権の必要な割り込みテーブルの操作を除去し、それを補填するよう Linux を改造することである。もう一つは割り込み発生時にカーネル動作中か判定する箇所をドメインでの実行に対応させることである。

Kerninst は x86 用実行時カーネル書き換えツールであるが、コードを挿入するには挿入先に十分な空きがある場合はフックとしてジャンプ (jmp) 命令を使

い、十分な空きが無い場合はブレークポイントトラップ (int3) 命令を使う。いずれの場合でもフックのところから挿入したコードに実行が遷移し、挿入コードと上書きした命令の実行後に元のプログラムにジャンプして復帰するようになっている。ブレークポイントトラップを挿入した場合はブレークポイントトラップのトラップハンドラの中でコードを挿入したために発生したトラップかを判断し、プログラムカウンタの値を挿入したいコードの先頭に書き換えて実行を再開する。

Kerninst は int3 命令のトラップハンドラである Linux の do_int3 関数を書き換えて int3 命令によるフックを実現する。Kerninst はこの書き換えを行うときに割り込みテーブル参照するが、これは特権命令であり、ドメインでは実行できない。Kerninst が割り込みテーブルを参照するのは通常の Linux カーネルでは do_int3 関数が static 関数であり、シンボルが解決できないからである。割り込みテーブルを参照する API を Xen は提供していないため、do_int3 関数をエクスポートするよう Linux のソースコードを改造し、Kerninst でもそれを利用するように変更した。

Kerninst はカーネル内でブレークポイントトラップが発生したときのみ動作する必要があるため、Kerninst を変更してこの判定を Xen 上のドメイン内のカーネルで動かしたときにカーネルモード実行を判定できるようにした。do_int3 関数はブレークポイントトラップが発生すると必ず呼ばれる関数である。そのため、ユーザーランドのプログラムがブレークポイントトラップを実行した場合でも呼ばれる。ユーザーモードかカーネルモードかの判定に用いているのが特権レベルを示すレジスタ (CR3 レジスタ) の値で、従来はこれがリング 0 の時にのみ挿入されたコードを実行するようになっていた。しかし、Xen 上の VM はすべてリング 1 で動作するため、この判定部分をリング 1 の時に挿入したコードを実行するよう変更する必要があった。

## 5. 実　　験

XenLASY の有効性を確認するために実験を行った。実験ではまず、マイクロベンチマークにより XenLASY のオーバーヘッドを測定して実用的なオーバーヘッドで XenLASY が動くことを確認した。次に、ケーススタディとして XenLASY を実際に使って調査を行った。なお、xflow 定義をコードに変換する部分は現在実装中であり、実験の際はルールに基づいて xflow 定義を手動でアスペクトに変換して実験を行っている。実験を行ったマシンは CPU は AMD Athlon™ 64 3500+ (2.2GHz) で、メモリーは 2GB (但し、ドメイン 0 に 256MB、ドメイン U に 128MB) であり、使用したソフト及びバージョンは Xen 3.0.4、CentOS 4.4 (Linux 2.6.16.33)、gcc 3.3.3、binutils 2.16 である。なお、ドメイン 0 もドメイン U も同じ OS、共通のカーネルを利用している。

**表 1** フロー ID 操作関数の実行時間 (ナノ秒)

| 関数名 | 実行時間 |
|---|---|
| get_new_flowid | 3 ± 0.0 |
| 空要素 get_flowid(empty) | 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 |

### 5.1 マイクロベンチマーク

まず、XenLASY で新たに加わった機能であるフロー追跡のための関数についてマイクロベンチマークを行った。ベンチマークでは各々の関数を 2 千回呼び出すのを 100 回行うのにかかる時間をタイムスタンプカウンタで計測し、その平均を求めた。なお、CPU のクロック周波数を動作中に変更するモジュールは動いておらず、実験中の CPU 周波数は一定である。

実験は新しいフロー ID を取得する get_new_flowid、フロー ID を登録する register_flowid、登録されたフロー ID を取得する get_flowid、フロー ID を消去する remove_flowid について行った。まず、新しいフロー ID を取得するのにかかる時間 (get_new_flowid) と空の状態で存在しない要素を取得するのにかかる時間 (空要素 get_flowid) を測定した。登録する構造体インスタンスのアドレスとフロー ID を 1 から 1 つずつ増やして登録した場合 (changeID) と同じフロー ID に異なる構造体インスタンスのアドレスを登録し続けた場合 (sameID) の 2 種類について、フロー ID の登録 (register_flowid)、参照 (get_flowid)、削除 (remove_flowid) にかかる時間を調べた。

実験結果は表 1 の通りである。同じフロー ID を使う場合もフロー ID が毎回違う場合も同じ処理時間がかかっている。どちらも登録に平均 33 ナノ秒かかり、削除に 32 ナノ秒しかかからない。また、データへのポインタからフロー ID を取得するのにかかる時間はいずれも平均 15 ナノ秒であり十分実用的な実行時間だと言える。

### 5.2 ケーススタディ

XenLASY を用いてドメイン U からドメイン 0 を介してネットワークに送出されるデータの流れを調査した。そして、この調査において I/O 処理フローが追

えることで、どの程度調査のためのリソース使用量を削減できたかを調べた。このときに用いたアスペクトは図 2 のようなもので、xflow 定義は図 3 と図 4 のようなものである。なお、xflow 定義からコードを作成する部分は現在実装中であるため、xflow 定義は手動で相当するアスペクトに変換して調査を行っている。

調査した結果は表 2 の通りである。経過時間は net/core/skbuff.c の 234 行目が実行されてからの経過時間をあらわし、ファイル名、行番号、関数名はそれぞれ上記のアスペクトにより保存したプログラムカウンタの値から調べたものである。これを見ると、ドメイン U のデータが tcp_make_synack で生成されて、ドメイン 0 にわたり、最後に SkGeXmit にてドライバからネットワークに送られ、それらのメモリーが開放されるまでを追跡できていることがわかる。この結果から、パケットを送るときにはドメイン 0 の netif_rx 関数で上位レイヤーに渡すための遅延キューに入れられてから取り出されて netif_receive_skb により上位レイヤーに上げる処理を行うまでの部分に時間がかかっていることがわかった。なお、ドメイン U からパケットの転送を VMM に依頼する関数である network_start_xmit 関数とドメイン 0 でそれを受け取る関数である netbk_fill_frags 関数の間での時間も取得できているが、上位レイヤーに上げる処理はこれよりも長い時間がかかっている。

次に、フローを追跡する機能があることでどの程度調査に必要なリソースを削減できているかを調べた。上記実験で用いたアスペクトでは get_flowid 関数を呼び、登録されていないものについては保存しないようにしている。これによるメモリー使用量削減、オーバーヘッドの低減の効果を調べるため、xflow ポイントカットによる制限無しでデータを無条件に保存するように書いたアスペクトを作り、ApacheBench で負荷をかけてデータ使用量を調べた。なお、保存するデータはアドバイス実行点のプログラムカウンタ、時刻、構造体インスタンスの先頭アドレスの 3 つである。この場合も 300 リクエストを 10 並列で行っている。

実験結果は表 3 の通りである。ドメイン 0、ドメイン U はそれぞれ各ドメインで使用したメモリー量を示している。フロー ID を用いて不要なデータを記録しないようにすることでフロー ID で峻別せずに記録していた場合に比べ全体で 60%程度のメモリー削減が出来ている。また、Apache のリクエスト処理性能はフローを使わない場合もフローを使った場合も毎秒 382 リクエストと特に差は無かった。

**表 3** フロー管理の有無によるメモリー使用量の違い (バイト)

| | ドメイン 0 | ドメイン U |
|---|---|---|
| フロー管理あり | 1,557,248 | 10,587,776 |
| フロー管理なし | 13,598,976 | 15,739,968 |

## 6. 関 連 研 究

### 6.1 アスペクト指向システム

データフローを選択するポイントカットとして、dflow ポイントカット[11] が提案されている。dflow ポイントカットでは、コールフローのようにデータフローを選択することが可能である。dflow ポイントカットでは複製を作った場合や選択された変数を用いて計算した結果についてもフローを追うことが可能である。しかし、dflow では、全自動でフローの追跡を行う処理のためにオーバーヘッドが高くなる傾向があり、xflow のようにプロファイリング用途には向かない。また、dflow ポイントカットはコンパイル時にフローを追うための情報を追加するシステムであり、XenLASY のように実行時にアスペクトを織り込むことは出来ない。また、dflow ポイントカットは Java で書かれたユーザーアプリケーションを対象としており、xflow ポイントカットは C 言語で書かれた Xen のドメイン上の OS カーネルを対象としている。

C 言語用の動的アスペクト指向システムには Arachne[6]、TOSKANA[7]、KLASY[16),17)] がある。Arachne はユーザーランドのプログラムを対象とし、TOSKANA は NetBSD カーネル、KLASY は Linux カーネルを対象とする。これらのシステムにはフローを追跡する仕組みが備わっていないので本論文で行ったような調査をするにはフロー調査のための仕組みを利用者が自分達で作る必要があり、不便である。KLASY は構造体のメンバーへのアクセスをポイントカットとして選択できるのでそれを使ってデータの流れを追えばフローを追跡することが出来そうにみえる。しかし、このシステムは Xen に対応しておらず、データフローを調べるためのポイントカットを持たないので調査のためにメモリーに収まりきらない膨大な量のログが出力されるという問題があり、実用的ではなかった。これに対し、XenLASY はフローを追跡する機能があるので必要なログだけを集めることが出来る。

分散環境を想定したアスペクト指向システムとして DAC++[1]、DJcutter[14] がある。これらのシステムはそれぞれ C++や Java で書かれたユーザーランドのアプリケーションを対象としているが、XenLASY は OS カーネルの挙動を調査するために VM をまたがってアスペクトの織り込みを行うアスペクト指向シ

**表 2** XenLASY によるトレース結果 (抜粋)

| 経過時間 ($\mu$s) | ドメイン | ファイル名 | 行番号 | 関数名 |
|---|---|---|---|---|
| 0.0 | U | net/core/skbuff.c | 234 | alloc_skb_from_cache |
| 3.4 | U | net/ipv4/tcp.c | 726 | tcp_sendmsg |
| 15.3 | U | net/ipv4/tcp_output.c | 495 | tcp_set_skb_tso_segs |
| 72.4 | U | net/core/dev.c | 1379 | dev_queue_xmit |
| 101.3 | U | net/core/skbuff.c | 471 | skb_clone |
| 130.8 | U | drivers/xen/netfront/netfront.c | 963 | network_start_xmit |
| 882.5 | 0 | drivers/xen/netback/netback.c | 1058 | netbk_fill_frags |
| 894.9 | 0 | net/core/dev.c | 1543 | netif_rx |
| 3310.4 | 0 | net/core/dev.c | 1730 | netif_receive_skb |
| 3332.7 | 0 | net/bridge/br_netfilter.c | 418 | br_nf_pre_routing |
| 3352.2 | 0 | net/bridge/br_forward.c | 70 | __br_forward |
| 3362.5 | 0 | net/bridge/br_netfilter.c | 760 | br_nf_post_routing |
| 3367.1 | 0 | net/bridge/br_forward.c | 35 | br_dev_queue_push_xmit |
| 3368.4 | 0 | net/core/dev.c | 1379 | dev_queue_xmit |
| 3462.6 | 0 | drivers/net/sk98lin/skge.c | 1416 | SkGeXmit |
| 3495.5 | 0 | drivers/net/sk98lin/skge.c | 1829 | FreeTxDescriptors |

ステムである。

### 6.2 プロファイラ

複数のモジュールやホストを横断して挙動を調査するプロファイラには MagPie[2),4)] や CauseWay[5)] がある。MagPie は実行時にログを出力して、あとからログを整理して原因を調べるシステムであり、CauseWay はメタデータを渡すことで一連の処理を追跡するシステムである。XenLASY はフロー ID という形でメタデータを保存するので CauseWay に似たシステムと言えるが、CauseWay と異なり通信時にメタデータを受け渡すための改造を OS のソースコードに加える必要が無く、織り込んでいない状態でのオーバーヘッドがない。また、XenLASY は任意のプログラムがアドバイスとして書けるため MagPie と異なり実行中に挙動を変えることも可能である。

Xen 用のプロファイラとして Xenmon[8)]、Xenoprofile[13)] が有名である。Xenoprofile についてはこれを使って実際に性能を向上した事例もある。[12)] これらのツールはいずれも特定のイベントの発生回数を調べ、イベントの発生回数が多いところに性能ボトルネックがあることを突き止めることができる。これに対し XenLASY は実際にプログラムのソースコードに則して調査するようになっているため、これらのツールで大まかに絞り、XenLASY で詳細を調べるという利用方法が考えられ、互いに相補的なツールであると考えられる。

## 7. ま と め

本論文では Xen 上で行われる I/O 処理を追跡するためのアスペクト指向システムである XenLASY を提案した。XenLASY ではドメインを越えたデータフローを追跡するアスペクトを簡単に書けるようにするため、xflow ポイントカットを提供している。VM を使うと I/O 処理に関連するモジュールが増えるため、原因の追求が難しくなる。しかし、xflow ポイントカットを用いてデータフローを追うと、各々のデータフローがどこで時間がかかっているかを調べられる。xflow ポイントカットを用いるとスレッドやドメインが変わってもデータの流れを追跡することが出来る。xflow ポイントカットの定義は始点、中継点、終点を明示でき、不必要な追跡を行わないようになっている。また、xflow ポイントカットの定義は名前をつけて利用し、再利用が出来るようになっている。ケーススタディではドメイン U から来たパケットをドメイン 0 で処理をするために上位レイヤーに送るところがボトルネックになっているとわかった。

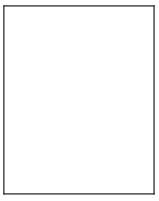
今後の課題としては、TCP などでデータが分割された場合にどのようにフロー ID を割り当てるか考える必要がある。同じフロー ID では区別できず、異なるフロー ID にすると関係がわからなくなるためである。また、現在の XenLASY ではドメイン間通信の際にヘッダのフィールドの空いている箇所にフロー ID を入れているが、空きが無い場合でも ID を渡せるように共有メモリーを用いることも出来るようにする必要がある。さらに、本システムは各ドメインの OS 内部にボトルネックがある場合は詳細に調べることが出来るが、VMM の内部などそれ以外の箇所にある場合は詳細に調べることが出来ない。I/O フローのボトルネックを調査する場合は、Xen が行う処理は単純であり、これはあまり問題はないと考えられる。今後は、実際のケーススタディでこれを確認する必要がある。また、汎用的なプロファイラとして使えるようにする

ため、Xen の内部のボトルネックを調査できるようにするのも今後の課題である。

## 参 考 文 献

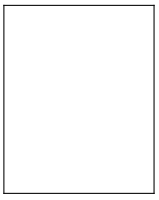

1) Almajali, S. and Elrad, T.: Coupling Availability and Efficiency for Aspect Oriented Runtime Weaving Systems, *Proc. of DAW05* (2005).
2) Barham, P., Donnelly, A., Isaacs, R. and Mortier, R.: Using magpie for request extraction and workload modelling, *Proc. of OSDI'04*, pp.18–18 (2004).
3) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *Proc. of SOSP '03*, pp.164–177 (2003).
4) Barham, P., Isaacs, R., Mortier, R. and Narayanan, D.: Magpie: online modelling and performance-aware systems, *Proc. of HotOS IX* (2003).
5) Chanda, A., Elmeleegy, K., Cox, A. L. and Zwaenepoel, W.: Causeway: Support for Controlling and Analyzing the Execution of Web-Accessible Applications, *Proc. of Middleware 2005* (2005).
6) 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*, pp.27–38 (2005).
7) Engel, M. and Freisleben, B.: TOSKANA: A Toolkit for Operating System Kernel Aspects, *Transactions on Aspect-Oriented Software Development II*, Vol.4242, pp.182–226 (2006).
8) Gupta, D., Gardner, R. and Cherkasova, L.: XenMon: QoS Monitoring and Performance Profiling Tool, Technical ReportHPL-2005-187, Hewlett-Packard Development Company, L.P. (2005).
9) Iyer, S. and Druschel, P.: Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O, *Proc. of SOSP'01* (2001).
10) Jones, S. T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R.H.: Antfarm: Tracking Processes in a Virtual Machine Environment, *Proc. of USENIX '06*, pp.1–14 (2006).
11) Masuhara, H. and Kawauchi, K.: Dataflow Pointcut in Aspect-Oriented Programming, *Proc. of APLAS'03* (2003).
12) Menon, A., Cox, A. L. and Zwaenepoel, W.: Optimizing Network Virtualization in Xen, *Proc. of the USENIX '06*, pp.15–28 (2006).
13) Menon, A., Santos, J.R., Turner, Y., Janakiraman, G.J. and Zwaenepoel, W.: Diagnosing Performance Overheads in the Xen Virtual Machine Environment, *Proc. of VEE'05* (2005).
14) Nishizawa, M., Chiba, S. and Tatsubori, M.: Remote pointcut: a language construct for distributed AOP, *Proc. of AOSD '04*, pp. 7–15 (2004).
15) Tamches, A. and Miller, B. P.: Fine-grained dynamic instrumentation of commodity operating system kernels, *Proc. of OSDI '99*, pp. 117–130 (1999).
16) Yanagisawa, Y., Kourai, K., Chiba, S. and Ishikawa, R.: A dynamic aspect-oriented system for OS kernels, *Proc. of GPCE '06*, pp. 69–78 (2006).
17) 柳澤佳里，光来健一， 千葉滋， 石川零：OS カーネル用アスペクト指向システム KLASY，情報処理学会論文誌:プログラミング， Vol. 48, No.SIG 10 (PRO33), pp.176–188 (2007).


(平成 19 年 8 月 01 日受付)
(平成 19 年 10 月 01 日採録)

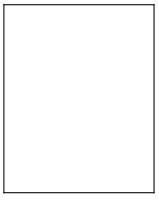
**柳澤 佳里**（学生会員）

2003 年東京工業大学理学部情報科学科卒業．2005 年同大学情報理工学研究科数理・計算科学専攻修士課程修了．オペレーティングシステム，言語処理系などの研究に従事．



**光来 健一**（正会員）

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．現在、東京工業大学大学院情報理工学研究科数理・計算科学専攻助教．博士（理学）．オペレーティングシステムの研究に従事．



**千葉 滋**（正会員）

1991 年東京大学理学部情報科学科卒業．1996 年同大学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准教授．博士（理学）．システムソフトウェアの研究に従事．