社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－OS－97　(5)
2004／8／1

通信処理のカーネル内競合を検出する アスペクト指向カーネルレベルロガー

柳澤佳里[†]　光来健一[†]　千葉滋[†]

本稿は、OS カーネル内での経過時間などのログを取得するために我々が開発したアスペクト指向システムについて述べる。本システムを使うと、カーネルソースコードを手動で変更せずに、カーネルの各所で網羅的にログ出力を行うことが容易に出来る。しかし、ログ出力を行う箇所を変えるたびに、カーネルの再コンパイル、再起動が必要では、実用上好ましくない。そこで我々は、アスペクトを実行する位置で割り込みを発生させるように実行時にカーネルコードを書き換える方法で、動的にアスペクトを組み込む方法を提案する。関数の呼び出し部分ではブレークポイント割り込み、変数の読み書き部分ではページフォールト割り込みを用いて実現する。こうして実現することで困難な基本ブロックの解析やデータ構造の解析の必要性が無くなる。この実装では負荷が大きいように見えるが実験の結果より通信処理に 300 程度のブレークポイント割り込みを挿入してもスループットが低下しないことが確認された。

An Aspect-oriented Kernel-level Logger for Finding Communication Bottlenecks

Yoshisato Yanagisawa,[†] Kenichi Kourai[†] and Shigeru Chiba[†]

This paper presents our aspect-oriented system for logging the time of events happening in an OS kernel. It enables to perform logging a large number of kernel events without modifying the source code by hand. However, previous implementation techniques have required recompiling the kernel and rebooting the system whenever the logged events are changed. We propose a new implementation technique for dynamically weaving an aspect into the OS kernel. In this technique, a software interrupt is issued when the program execution reaches the join points where an aspect should be executed. Our system substitutes a machine instruction for software interrupt for an original instruction if the join point is a function call. It uses a page-fault interrupt if the join point is memory access. Our technique can be easily implemented since it does not need complicated analysis of basic blocks and data structures. Although this technique might seem to involve heavy execution overheads, our preliminary experiments showed that the overheads do not significantly decrease the throughput if less than 300 software interrupts are inserted through the execution path for network communication.

1. はじめに

大容量のファイルをダウンロードしている状況でネットワークゲームなどの小さいデータを頻繁にやり取りするアプリケーションを使っているとその動きが一瞬遅くなる場合がよくある。この原因にはネットワーク上でのデータ送信の競合の他に OS カーネル内部での競合も考えられる。OS カーネル内部での競合として、例えばネットワーク内でパケットが失われてしまったために再送が発生しており、再送のための処理を何度も行うことでシステムの負荷を上げて性能低下を引き起こしている場合がある。また、大量のパケット受信処理に CPU を取られてしまいネットワークにパケットを送信するための処理が進まない場合も考えられる。

このような問題を解決するにはまず OS カーネル内でどのような競合が起きているかを調べるために各処理の経過時間を測定する必要がある。しかし、ユーザーランドから測れる精度も箇所も限られている。カーネル内部で競合を調べるためツールとしてデバッガがあるが、デバッガでは内部状態を調べたり時間を取得したりするときに停止を伴うので経過時間の測定には使えない。

この他の手段として手作業で経過時間のログを取得するコードを組み込む方法がある。しかし、この方法

† 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

ではログを取るべきところでログを取るコードを挿入するのを忘れてしまう恐れやログを取り終わった後にコードを削除し忘れてしまう恐れがある。また、利用者がログを取りたい箇所は測定結果の内容によって変わることが考えられるが、その度にカーネルの再コンパイル、再起動を繰り返すのは時間がかかる上、OSカーネル内部で管理しているキャッシュやファイルディスクリプタなどのデータが失われるので避けたい。

我々はこの問題を解決するためのアスペクト指向システムを提案する。このシステムでは、カーネル内部で関数が実行された時、関数が呼び出された時、変数参照が実行された時、前後に、指定したコードを暗黙のうちに呼び出すように指定できる。そのようなコードのことをアスペクトと呼ぶ。本システムでは、ログ取得用のコードをこのアスペクトとして記述する。アスペクトはカーネルのソースファイルとは別に記述するため、ログを取り終わった後でコードを取り忘れることもない。また、入出力キューの操作はカーネルコード内の多数の場所で実行されるので、どこでログ出力を行うかの指定が容易ではないが、入出力キューを指す変数に対する参照が実行されたときに、ログ取得用のアスペクトを実行するように指定できるので、従来方法よりも容易であると考える。

我々はカーネルの再起動を避けるために、アスペクトを実行時に組み込み、取り外しできるようにする。その実現には割り込みを用いる。関数の先頭や関数呼び出しの位置でのアスペクト実行はその位置の機械語命令をブレークポイント割り込み命令に置き換えることで実現する。実行がその位置にくると割り込みが発生し、割り込み処理ルーチン中で対応するアスペクトの実行を行う。一方、変数アクセスの位置でのアスペクトの実行については仮想メモリの保護機能を利用する。指定した変数がアクセスされるとページフォールトが起こり、割り込みルーチン中で対応するアスペクトを実行する。

我々は以上のようなシステムの性能を見積もるための予備実験として FreeBSD 5.2.1R-p8 上でブレークポイント割り込み命令およびそれに関わる一連の処理にかかる時間と関数呼び出し命令の実行にかかる時間の比較を行った。その結果、ブレークポイント割り込み関連の実行にかかる時間は関数呼び出しの実行に比べてかなり大きいが、システムコールを 1 度呼ぶのに比べるとはるかに小さいとことがわかった。また、同 OS 上で `ip_output` 関数にブレークポイント命令を挿入し、その実行数とスループットとの関係を調べたところ、ブレークポイント命令の実行数が 300 個以下ならばスループットへの影響はほとんど無く、400 個以上ではスループットが低下することがわかった。

以下、2 章では、カーネルレベルロガーの必要性とそれに対する要求について詳しく述べる。3 章では、我々のアスペクト指向システムの実現方法について述べる。4 章では、我々が行った実験について述べる。5 章で関連研究に触れ、6 章で本論文をまとめて、今後の課題を述べる。

2. カーネルレベルロガーの必要性

対話的な通信処理と大容量のデータを送るような処理を同時に行う場合、対話的な処理の反応が遅くなってしまう場合がある。例えば、大容量ファイルをダウンロードしている状況でネットワークゲームをしているとゲームのキャラクターに移動を指示してから実際にそれが画面に反映されるまでにしばらく待たされることがある。このような遅延の原因はカーネルおよびネットワークにてネットワークゲームの通信と大容量ファイルのダウンロードの通信が競合していることだと考えられる。よって、この問題を改善するにはカーネル内部で競合が起きていないか、起きているならどこで起きているのかを調べる必要が出てくる。

カーネル内の競合を調べるには通信処理の実行時間などの詳細なログが必要である。通信処理についてカーネル内部でログを取る場合には次のような箇所で調べるのが妥当であると考えられる。

(1) `write` システムコールからデバイスドライバまで
(2) デバイスドライバから `read` システムコールでまで
(3) 再送処理

(1) を調べることでカーネルがユーザープロセスからデータを受け取ってからネットワークにデータを送信するまでの箇所で競合が起きているか調べることができる。(2) を調べることでネットワークからくるデータをデバイスドライバが読み込んでから実際にユーザープロセスに届くまでにカーネル内部で競合が起きているかを調べることができる。さらに、(3) を調べることでネットワークに送信した後の競合の状態を調べることができる。

カーネル内部で経過時間などのログを取る方法として、従来はログ出力のための処理をカーネルソースコードに書き込む方法が使われてきた。この手法ではログを取る位置を変えるとソースコードの変更、再コンパイルマシンの再起動が必要になるがこれは避けるべきである。この理由としてまず再起動には多大な時

間がかかることがあげられる。再起動では全てのサービスを停止させた後、計算機の状態のチェックやファイルシステムの整合性のチェックを行ってから全てのサービスを再開させるため、通常のサービスを停止、再開するのに比べて長い時間がかかる。また、OS の再起動中に全てのサービスが停止するため、サーバの場合は利用者に迷惑をかけることになる。さらに、再起動を行うとカーネル内部のキャッシュの状態や接続しているネットワークの状態が大きく変化する可能性がある。このような変化は問題の再現を難しくする。例えば、何らかのネットワークアクセスが来ているときに再起動すると相手のホストの状態に変化が生じるので、同じ状況が次にいつ再現するかは分からない場合が多い。

また、ログを取る処理をプログラムに手動で入れるため、入れ忘れや取り忘れが生じる恐れがある。入れ忘れが生じてしまうと必要な箇所でログ出力を得ることができず、競合が起きているかを判断するのに必要なデータを一部得られない箇所が出てくる恐れがある。入れたログを取り忘れるとシステムの実行中に性能低下を引き起こしたり、システムの秘密情報を出力している場合には潜在的なセキュリティーホールとなったりする恐れがある。

動的にログを取る位置を変えられる方法として、ログを取るコードを予めカーネル内部に入れておき、必要に応じて on/off するという方法がある。この方法ではあらかじめログを取ると予測した範囲にフラグとログ出力のコードを埋め込んでおく。しかし、この方法では必要なログ出力が得られない可能性がある。競合を探すためにログ出力をする場合にはどこでどのようなログを取ればよいか分かっていることは少ないと考えられる。出てきたログに応じて取るログの内容や箇所を変えたい場合に対応できない。逆に、あらゆる箇所で様々なログ出力を選べるようにして自由度を上げようとすると、その場で出力できるものの数だけフラグを作らなくてはならなくなる。この場合、フラグが大量に作成されて実行時オーバーヘッドが大きくなり、カーネルのサイズも増大してしまうという問題が生じる。

3. カーネル用アスペクト指向システム

このような問題を解決するカーネルレベルロガーを作るために、我々はカーネル用のアスペクト指向システムを提案する。これは関連したログ処理と挿入位置をまとめてアスペクトに記述したものを実行時にカーネルに適用するシステムである。このシステムを用いることで以下が可能になる。

- **自動化** 関連するログ処理をアスペクト単位でまとめて挿入削除でるため、取り忘れがない
- **自由度** あらゆる関数実行、関数呼び出しの前後やあらゆる変数アクセスの箇所にログ処理を挿入できる
- **無停止** アスペクトを実行時に挿入、削除できるため、マシンの再起動が必要ない

3.1 アスペクト指向とは

アスペクト指向とはロギングや先読み、同期処理のような様々なモジュールに跨る関心事（crosscutting concern）をアスペクトいう概念を用いてモジュール化し記述するという考え方である。このアスペクト指向にはジョインポイント、ポイントカット、アドバイスという 3 つの重要な要素がある。

ジョインポイント

ジョインポイントとは関数呼び出しや変数アクセスなどプログラムの実行フローの中で定義されたポイントのことである。例えば、関数呼び出しのジョインポイントは関数を呼び出す時と呼び出しから復帰する時を表すポイントである。このジョインポイントの有効期間は関数呼び出しが開始してから復帰するまでであるが、ジョインポイントの表すポイントは呼び出しの開始または終了の瞬間に限定される。

ポイントカット

ポイントカットとはこのジョインポイントを指定する方法のことで、プログラム中のすべてのジョインポイントから特定のジョインポイントを抽出するのに使われる。このポイントカットを指定する記述をポイントカット指定子と呼ぶ。このポイントカット指定子には次のようなものがある。

- **call ポイントカット指定子** 関数呼び出しの箇所を抽出する
- **execution ポイントカット記述子** 関数の実行箇所を抽出する
- **set ポイントカット記述子** 変数の値が変更される箇所を抽出する
- **get ポイントカット記述子** 変数の値が参照された箇所を抽出する

例えば、次のポイントカット指定子は int 型関数 `ip_output` あるいは void 型関数 `div_input` のどちらかの関数呼び出しを抽出し、`diverting` というポイントカットと名付けている。

```
pointcut diverting():
    call(int ip_output(...))
        || call(void div_input(...))
```

これらポイントカット記述子でカーネル内部のジョインポイントを指定することを考える。すると、set/getポイントカット記述子はデータの流れのログを取るのに使える。具体的には調査したい mbuf 構造体をset/get ポイントカットで指定してそのアドバイスにて現在の mbuf 構造体の状態や実行している関数、時間を出力することで行う。

アドバイス

アドバイスはポイントカットで指定したジョインポイントで実行されるコードを定義する。アドバイスには次のものがある。

- **before アドバイス** プログラム実行がジョインポイントに到達する直前で実行される
- **after アドバイス** プログラム実行がジョインポイントから復帰した直後に実行される
- **around アドバイス** プログラム実行がジョインポイントに達したときに実行される

例えば、次のアドバイスは先の例で `diverting` が `ip_output` 関数あるいは `div_input` 関数が呼ばれたときを示すポイントカットとして定義されているので、これらの関数が呼ばれた直後に簡単なメッセージを表示する。

```
after() : diverting() {
    <code to print message>
}
```

上記で述べたようなポイントカットとアドバイスをあわせたものをアスペクトと言う。そして、このアスペクトを実際のプログラムに合成する処理を weave という。逆に合成したアスペクトを取り除く作業を unweave という。

3.2 call/execution ポイントカットの実現方法

実行時に weave できるようにするために call/execution ポイントカットはブレークポイント割り込みを発生させることで実現する。これはポイントカットで指定した関数や関数呼び出しの位置の命令をブレークポイント割り込み命令（`int3`）に書き換え、その割り込み処理ルーチンにて書き換えた箇所にあった命令と対応するアドバイスを実行するというものである。図 1 はこの流れを示したものである。実線はプログラム中のジョインポイントからアドバイスの実行に至る道筋を示しており、点線はアドバイスを実行した後にプログラム中に戻る処理が通る道筋を示している。この図では関数 `ip_output` が呼ばれるたびにブレークポイント割り込みが発生し割り込み処理ルーチン `trap` が呼び出される。そして、割り込み処理ルーチンからはブレークポイント割り込みであることを調べた後に

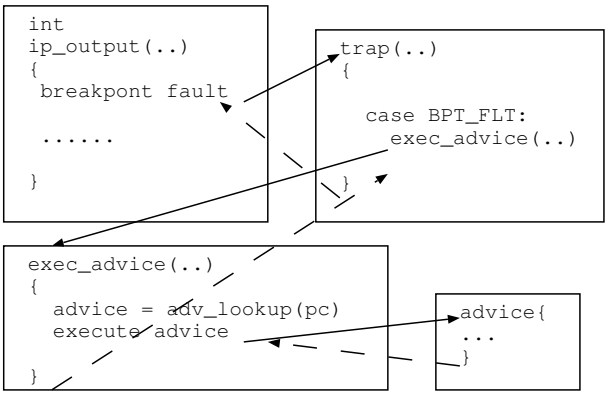

図 1 call/execution ポイントカットでのアドバイス実行手順

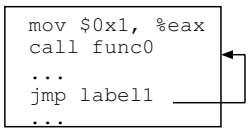

図 2 jmp 命令による実装がよくない例

`exec_advice` 関数が呼び出される。この関数では割り込みが発生した点のプログラムカウンタをキーとしてテーブルを引くことで実行されるべきアドバイスを探す。そして、そのアドバイスを実行し、点線の経路をたどってそれまで実行していたプログラムに戻る。ブレークポイント割り込みを用いた実現方法ではポイントカットで指定した点以外には変更を行わないのでその点以外は従来通りの実行速度を保つことができる。

`jmp` 命令や `call` 命令でアドバイスに飛ばすことで call/execution ポイントカットを実現することも可能である。しかし、`call` 命令も `jmp` 命令も 5 バイト命令であり、ブレークポイント割り込み命令と違って 1 バイト命令ではない。このため、現在ある命令を書き換えるという実装では、書き換えた箇所が `jmp` 命令や `call` 命令の飛び先になっている場合に正常動作時に実行されるのとは違った命令が実行される危険性が出てくる。例えば、`mov` 命令（2 バイト命令），`call` 命令（5 バイト命令）と連続していて、後ろの `call` 命令の箇所（label1 と名付ける）に `jmp` 命令で飛ぶようなプログラムがあったとする（図 2）。この場合、ポイントカットを実現するために `mov` 命令の箇所を `jmp` 命令で上書きすると call 命令も上書きしてしまうことになる。そして、この書き換えられた `call` 命令があった箇所は label1 への `jmp` 命令の行き先であったので label1 への `jmp` 命令が発行された後にどのような動作をするかは保証できない。 このような不具合を

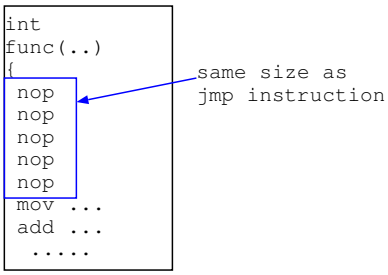

図 3 ジョインポイントの糊代部分を nop 命令で作成する例

避けるためには書き換える箇所が基本ブロック☆の範囲内に留まっているか否かを調べなくてはならない。この操作はプログラムの実行フローを調査する必要があり weave に時間がかかるため、実行時に weave を行うシステムでは問題である。

上記の問題を解決するために jmp 命令や call 命令でアドバイスを挿入するための糊代となる部分を作っておく実装も考えられる。[6] 具体的には全てのジョインポイントに jmp 命令や call 命令の大きさの nop 命令をあらかじめ書いておき、weave される場合にはこの箇所の命令を所定の jmp 命令や call 命令に書き換えるという実装である（図 3）。この糊代は基本ブロック内なので jmp の飛先になることは無い。しかし、この実装ではポイントカットできる箇所を増やすとカーネルのサイズが増大する。これは使えるディスク容量やメモリ容量が小さいシステムで使うと本システムが容量不足を引き起こすことを意味する。また、糊代の部分のための大量の nop 命令で動作が遅くなる恐れがある。ポイントカット可能なジョインポイントの数を限定すれば、この方法によるオーバーヘッドは小さくなるが、ポイントカットできる箇所の自由度が減るので、そのような対策は避けるべきである。

3.3 set/get ポイントカットの実現方法

実行時に weave できるようにするために、set/get ポイントカットはページフォールト割り込みを発生させることで実現する。この実現方法では仮想メモリの保護機能を使い、ポイントカットする変数が入っているページを set ポイントカットなら書込禁止、get ポイントカットなら読込禁止に設定する。そして、アクセスしたときにページフォールト割り込みを起こさせる。設定したページのページフォールト割り込みが起きた場合はその割り込み処理ルーチンにてそれに応じたアドバイスを実行する。図 4 は set(m) というポイントカットを行った場合のアドバイスの実行時の動作

☆ 分岐命令を持たず途中から合流もされない命令列

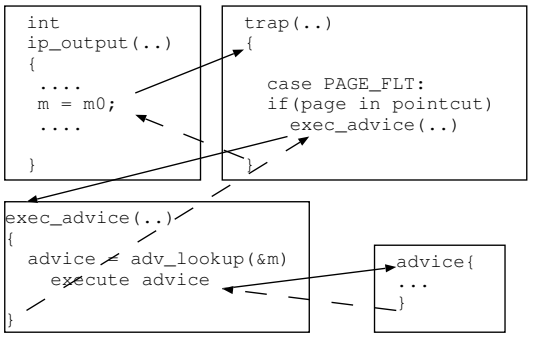

図 4 set/get ポイントカットでのアドバイス実行手順

を示している。実線はアドバイスを呼び出すまでの動きで、点線はアドバイス実行後の動きを示している。ここでは m を set ポイントカットで指定しているため m に書き込んだ際にページフォールト割り込みが発生し、trap 関数のページフォールト処理へと実行が移る。そして、ページフォールトが設定した変数の箇所で発生していたら exec_advice 関数を呼び出す。この関数では変数 m のアドレスからアドバイスを調べ、対応するアドバイスを実行する。アドバイス実行後は点線の経路をたどってそれまで実行していたプログラムに戻る。

この方法では同一ページに存在する他の変数へのアクセス時にも割り込みが発生するため、オーバーヘッドが問題となる。しかし、実際にはそれほど大きなオーバーヘッドにはならないと考えられる。カーネルでは数バイトの大きさしかないプリミティブ変数への参照ををポイントカットする機会よりもサイズの大きい構造体への参照をポイントカットする場合が多いと考えられる。また、カーネルでは一括して構造体のためのメモリを確保することが多いので同一ページ上のデータはすき間にプリミティブ変数が入る形にはならずページが同じ構造体で埋められている場合が多い。実際、mbuf は連続で取られその容量は 256 バイトなのでページあたり 16 個しかない。そのため、構造体をポイントカットするときにはページフォールトを発生させる変数の数は比較的少なく、オーバーヘッドを抑えられると思われる。

set/get ポイントカットの実現方法としては指定した箇所にアドバイスを呼び出す jmp 命令やブレークポイント割り込み命令を挿入するという方法も考えられる。しかしながら、この方法ではポインタ変数経由で指定した変数に間接的にアクセスされる場合に対応するのが困難である。このようなポインタ変数によるアクセスに対応するためには weave 時に set/get ポ

```
aspect log_network_in {
  pointcut network_in():
    call("% ip_input(...)||% tcp_input(...)");
  advice network_in(): before() {
    printf("In Network Input Function.");
  }
}
```

図 5 アスペクト記述の例

イントカット記述子で指定された変数のアドレスを持つ可能性があるポインタ変数についても調べそこにも jmp 命令やブレークポイント割り込み命令を挿入する必要がある。しかし、ポインタ変数は機械語中では通常の変数と同じ様に扱われるので静的な調査で調べあげることは出来ない場合が多い。さらに、実行時に調べる場合には全てのメモリアクセスを調べなくてはならないため調べること自体が多大なオーバーヘッドになる。

3.4 weave/unweave の実現方法

上述のようにアスペクトを実行時に動的にカーネルに weave/unweave するシステムを構築するにあたって、人間にわかりやすいアスペクト記述から実際にカーネル内部で使う形式へとどのようにして変えるかやその変化をカーネル内部とユーザーランドとのどちらでやるかを考察した。

ユーザーの書くアスペクト記述は使いやすさを考慮し、AspectC[1] 風に書けるようにする。例えば、図 5 のように書くと返り値や引き数の型や数に関係なく ip_input あるいは tcp_input という名前の関数が呼ばれる点が network_in と名付けられたポイントカットとして抽出され、このポイントカットの before アドバイスとして printf("In..."); が入っているようなアスペクト log_network_in が定義される。

次に、上記のような AspectC 風の言語で書かれたアスペクトをテキスト形式でカーネルに渡し、カーネル内部でコンパイルを行う。一方、コンパイルしてからカーネルに渡すことも考えられる。一般的な Unix システムでは実行時にカーネルの再構築が出来るので kernel のバイナリファイルを見て変数の格納されるアドレスの位置を調べ、アドバイスのコンパイルを行った場合に動いているカーネルに正しく適用できないコードが作られる可能性がある。この危険性を考え、カーネル内部でアスペクトをコンパイルするという設計を採用した。

カーネル内部ではアスペクトをコンパイルして次の情報を持つ。

- ポイントカット指定子
- アドバイスをコンパイルしたバイナリ

この情報をもとに次の手順で weave を行う。

(1) ポイントカット指定子で指定された箇所のアドレスを調べる
(2) call/execution ポイントカット指定子で指定した箇所のアドレス、set/get ポイントカット指定子で指定した変数のあるメモリアドレスをキーにしてアドバイスを登録する。
(3) call/execution ポイントカットの位置にブレークポイント割り込み命令を書き込む。上書きされる命令は保存しておく。また、set/get ポイントカットの位置の仮想メモリの保護属性を変更する。

unweave 時には call/execution ポイントカットの場合はブレークポイント割り込み命令を元の命令に戻し、set/get ポイントカットの場合は仮想メモリの保護属性を元に戻す。

4. 実　　験

我々のシステムでは割り込みによりアドバイスを weave するようにしているがそれが OS の性能にどの程度の影響を与えるのか判断するために実験を行った。

4.1 割り込みと関数呼び出しの実行時間の比較

まず、関数呼び出し命令（call）とブレークポイント割り込み（int3）との実行時間の比較を行った。この比較は我々のシステムを関数呼び出しにて実装した場合と比べてどの程度性能劣化が起きるかについて定量的な考察を加えるために行った。実験は 10000 回行い、その中の最小値を比較することとした。なお、実験環境は CPU AMD Athlon XP 2200+（クロック周波数 1.8GHz）でメモリが 1GB、OS は FreeBSD 5.2.1-p8 である。

実験に際しては正確さを期すために実験を行うためのシステムコールを用意してカーネル内部に制御を移し、カーネル内部で mtx_lock(9) 関数を用いジャイアントロックを行って割り込みを禁止した上で測定することとした。具体的には次の手順で測定を行った。

(1) 割り込みを禁止
(2) 関数呼び出しあるいはブレークポイント割り込み前の時刻の測定
(3) 関数呼び出しあるいはブレークポイント割り込み
(4) 関数呼び出しあるいはブレークポイント割り込み後の時刻の測定
(5) 割り込みを許可

そして、手順2と4との時間差により関数呼び出しあるいはブレークポイント割り込みにかかった時間を算定する。なお、関数呼び出しでは別のファイルに記述された中身がない関数を呼び出し、ブレークポイント割り込みでは汎用割り込み処理関数の sys/i386/i386/trap.c の `trap` 関数中でデバッガ起動ルーチンである `kdb_trap` 関数を呼び出す箇所の直前でリターンして上記の実験を行った。

時間の測定は正確さを期すために CPU のクロックに連動して 1 ずつ値が増える TSC（Time Stamp Counter）を用いて測った。この CPU の場合だと 1.8GHz のクロック周波数なので 1 秒あたり 18 億回カウンタが増える計算になり、1 クロックあたり約 0.56 ナノ秒である。

4.1.1 実験結果

実験の結果、ブレークポイント割り込みには最小値で 373 clock かかり関数呼び出しには最小値で 15 clock かかることがわかった。この結果から呼び出しと終了にかかる時間は関数呼び出しの方が各段に早いことがわかる。しかしながら、比較的軽いシステムコールである `getpid(2)` ですら 28596 clock かかることを考えるとブレークポイント割り込みの負荷はさほど大きくはないと考えられる。

4.2 割り込み挿入時のネットワークの性能低下の調査

4.1 節の実験結果を踏まえ、ブレークポイント割り込みが入ると通信処理にどの程度性能劣化が起きるのかを調べる実験を行った。実験では IP パケットにヘッダを付け下層レイヤーに送る `ip_output` にてブレークポイント割り込みを任意の回数起こし、そのブレークポイント割り込みが起きた回数とスループットの関係を測定した。

実験では 2 台の計算機を 100Mbps のスイッチングハブを介して接続した。そして、その 2 台の計算機をサーバーとクライアントに分け、クライアントからサーバーへ 1454bytes のデータを 10240 回 `write(2)` を使って送りその前後でかかった時間を `clock_gettime(2)` にて測定した。そして、このかかった時間で送信したデータ量を割ることでスループットを計算した。ブレークポイント割り込みを起こす回数を 0 から 1000 まで 100 刻みで増やしていったときにこのスループットがどのように変化するかを調べた。なお、実験環境はサーバー、クライアントとも CPU AMD Athlon XP 2200+（クロック周波数 1.8GHz）、メモリ 1GB、NIC Intel Ethernet 100/Pro であり、サーバー側の OS は FreeBSD 4.10R でクライアント側の OS は FreeBSD 5.2.1R-p8 であった。

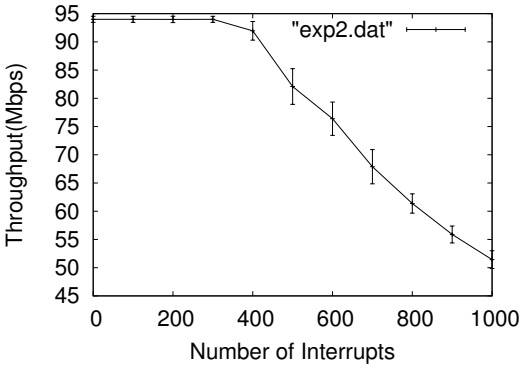

図 6 割り込み回数の増加とスループットの関係

4.2.1 実験結果

実験の結果は図 6 の通りである。この図では横軸にブレークポイント割り込みの回数をとり、縦軸にスループット（Mbps）をとっている。 このグラフからブレークポイント割り込みの回数が 300 回以下の場合はスループットは割り込みの影響をほとんど受けないことが読み取れる。そして、割り込みの回数が 400 回以上のところから劇的な性能の低下が起きはじめることがわかる。

5. 関連研究

AspectC[1] および AspectC++[2),7)] は C 言語を AOP 拡張するために作られた汎用的なアスペクト指向システムである。この AspectC を用いて OS カーネルの実装をすることで OS カーネルのソースコード中に散らばりがちな先読み処理ををまとめてモジュール性を向上させるという試みがなされている。[4),5)] しかしながら、AspectC および AspectC++はプログラムのコンパイル時にアスペクトを weave するシステムであり、そのアスペクトを変更するにはコンパイルしなおす必要がある。そのため、ログを取る箇所を実行時に試行錯誤しながら調べるということには不向きである。また、コンパイルしなおした場合には再起動する必要があり、場合によっては問題となった現象が再現するのに長い時間がかかる可能性もある。

μDyner[6] は実行時にアスペクトを weave できるシステムの一つである。μDyner は before アドバイスへ飛ぶ `jmp` 命令を書き込むための余白をコンパイル時に各関数の頭に埋め込む。そして、アスペクトの weave はアスペクト記述の書かれたコードをメモリ中にロードし、関数の頭の箇所をそのアスペクトへと至るコー

ドに書き換えることで行う。unweave は jmp 命令を元に戻し、ロードしたコードを削除することで行う。μDyner はコンパイル時に jmp 命令を書き込むための余白を作成するためこの箇所しかジョインポイントにできない。よって、これの自然な拡張でジョインポイントを増やそうとするにはそれらの地点に jmp 命令を書き込めるだけの余白を作成する必要が出てきてコードサイズの増大につながる。また、カーネルコンパイル時にどこにアスペクトを weave するか決めてしまうのでアスペクトを weave できる箇所の自由度が失われてしまう。

LKST[3),8)] は Linux カーネルで発生したイベントをトレースすることができるシステムである。そして、これはプロセスのコンテキストスイッチやシグナルなどカーネル内部で発生した様々なイベントをトレースすることができる。また、イベントのトレースを行う関数の代わりに任意のコードを実行させるようにすることもでき、イベントをジョインポイントとしたある種のアスペクト指向システムととらえることもできる。しかしながら、LKST ではイベントハンドラのみしかポイントカットできず例えばネットワークに纏わるコード全てで経過時間をトレースしたいというような利用方法には向かない。

6. まとめと今後の課題

ネットワーク I/O 処理のプロファイリングを行い、OS の改善を目的としたカーネル レベルの実行時アスペクト指向システムを提案した。割り込みにより実行時にアスペクトを埋め込む実現方法について議論した。その結果、call/execution ポイントカットはブレークポイント割り込みで実現し set/get ポイントカットは仮想メモリの保護機能を使うことで実現するのが妥当であると考えた。さらに、実験によりブレークポイント割り込みの負荷は関数呼び出しに比べて大きいもののその量が少ない場合にはネットワークの主要な関数に埋め込んでもネットワーク I/O のスループットはさほど変化しないことを確認した。

今後の課題は次の通りである。

- 本システムの実装
- 構造の変化を追求できるようにする拡張
- 本システムを用いて競合要因の調査
- 判明した競合要因を元にカーネルの改良

まず、論文中で述べたようにカーネル内でアスペクトをコンパイルし、weave、unweave を行えるシステムを作り上げる。次に、実際のネットワーク I/O ではデータ配列から mbuf 構造体のチェーンへとデータ構造が大きくかわるところがあるのでその構造を追っていけるように拡張したいと考えている。さらに、これらのことを行った上でカーネル内のネットワーク I/O の競合の要因を調べ、カーネルの改良をしたいと考えている。

参 考 文 献

1) http://www.cs.ubc.ca/labs/spl/projects/aspectc.html.
2) http://www.apectc.org/.
3) http://lkst.sourceforge.net/.
4) 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.
5) 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.
6) 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.
7) 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.
8) 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.