システムソフトウェアと
オペレーティング・システム
（２００１．　６．２８）
87－4

IEEE 1394 を利用した OS プロファイラの開発

山之内 暢彦　多田 好克
電気通信大学大学院情報システム学研究科

プログラムの動作を把握、測定するのに、デバッガやプロファイラがしばしば用いられる。しかし、測定対象が OS の場合は問題があり、(1) デバッガを使って動作を止めると、リアルタイム性が失われ、本来とは異なる動作を測定してしまう、(2) 従来のプロファイリング手法では、カーネル内の割り込み禁止区間の動作を測定できない。それらの問題を解決するため、我々は IEEE 1394 を利用した測定法を提案する。IEEE 1394 を使うと、リアルタイム性を損なうことなく OS の動作状態を取得することができる。本稿では、プロセス等の動作についてリアルタイムに測定を行い、手法の有効性を確認した。また、その手法を応用し、プローブ効果の低い OS プロファイラを開発した。

Developing an Operating System Profiler Employing IEEE 1394

YAMANOUCHI Nobuhiko and TADA Yoshikatsu
The Graduate School of Information Systems,
The University of Electro-Communications

The actual behavior of programs can be observed through a debugger or a program profiler. To measure operating systems behaviors, however, there are some problems: (1) observed behavior can be misled since debugger trace spoils realtimeness, (2) with traditional profilers, nothing is observed while interrupts are disabled. To overcome these problems, we propose a method of measurement employing IEEE 1394, high speed serial bus standards. This enables us to know operating system's behavior without loss of realtimeness. In this paper, we show that our method can illustrate interactions among processes in realtime. And applying our method, we developed the profiler which is able to measure operating systems with low probe effect.

1 はじめに

ネットワークが急速に高速化している。それに伴い、通信路のスループットより、むしろ OS カーネル内の処理がボトルネックになってきた。とりわけスケジューリングやネットワークに関する処理が問題となる場合が多い。これらの性能を改善するにあたり、まずはカーネルの動作を把握、分析することが重要である。

プログラムの動作を把握する手法には様々なものがある。それには、printf を使った状態出力、デバッガによるトレースといった簡便なやり方がある。しかし、これらの方法では動作のリアルタイム性が失われるため、OS カーネルの場合、誤った測定結果を得ることがある。

たとえば、ネットワーク処理はほかのコンピュータや通信路との相互動作によって行われる。そのため、リアルタイムで測定しなければ本来とは異なる動作を測定する可能性が高い。printf のオーバヘッドによって動作が遅れたり、デバッガを使って動作を止めることは避けなければならない。

一方、動作を把握する手段として、プロファイラが用いられることも多い。しかし、これにもリアルタイム性の問題があるほか、カーネル内の動作を測定できないことがある。

多くのプロファイラは、プロシージャやブロックの実行回数、実行時間等を出力する。これらの出力結果を得るためにプロファイラはタイマ割り込みを使用するが、この方法では割り込み禁止区間の処理を測定できない。OS カーネル内には割り込み禁止区間が数多く存在するため、不十分な測定結果を得ることになる。

本研究では、IEEE 1394 [1] ホストコントローラの機能を使い、一方のコンピュータから被測定コンピュータの動作状態を「覗き見」することを行った。このとき、被測定側の動作を止めないため、リアルタイム性を失うことがない。また、割り込み禁止や動作モード等、CPU の状態にかかわりなく覗き見することができる。

今回はこの手法を使い、プロセスの動作状態、割り込みの状態等をリアルタイムに取得、図式化した。また、覗き見の手法を応用し、IEEE 1394 を利用したカーネルプロファイラを開発した。

従来のプロファイラでは、測定データの収集・保存処理が影響して、リアルタイム性が損なわれる。一方、開発したプロファイラでは、測定データの取り扱いに覗き見の手法を応用し、リアルタイム性を最大限確保した。

以下、**2 章**で IEEE 1394 の特徴、覗き見の手法について説明する。**3 章**では、実装について述べたあと、IEEE 1394 の通信性能について評価する。そして、**4 章**で覗き見を利用した実際の測定例を示す。**5 章**では、開発したプロファイラの詳細について説明する。

2 IEEE 1394 による動作状態の取得

本研究では 2 台のコンピュータを用い、その間を IEEE 1394 で接続する。一方は被測定コンピュータで、測定対象の OS やユーザプログラムを動作させる。もう一方では測定用プログラムが動作し、測定の制御、IEEE 1394 を介した動作状態の取得、ハードディスクへの記録を行う。なお、動作状態の取得には、IEEE 1394 のコントローラが持つ Physical Read 機能を使用する。

2.1 IEEE 1394 の特徴

高速シリアルバス規格の IEEE 1394（FireWire、i. Link とも呼ばれる）は、公称 400Mbps のスループットを持つ通信手段である。通信モードには、自動的に誤り検出、パケット再送を行う Asynchronous モードと、映像等の通信に適した Isochronous モードの二つがある。前者は TCP/IP over IEEE 1394 や記憶装置との通信に用いられ、本研究ではこのモードだけを使用する。

通信パケットヘッダには 64 ビットのアドレスがあり、上位 16 ビットが通信ノード等の識別、下位 48 ビットがノード内のアドレスを指定する。

2.2 Physical Read 機能

IEEE 1394 のコントローラには Physical Read と呼ばれる機能がある。これは、通信先コンピュータのメインメモリを読み取ることができる機能である。具体的には、以下の順に処理が行われる。

1. メモリ内容を取得したいコンピュータが通信先に Read Request パケットを送る。
2. それを受け取ったコントローラは、パケットヘッダ中のアドレス（64 ビット中の下位 32 ビット）を物理メモリアドレスとして解釈し、メモリか

ら内容を読み取る。

3. 読み取った内容から Read Response パケットを構成し、送り返す。

通常の NIC と違い、Read Request パケットの処理はすべてコントローラが行うため、CPU に割り込みが発生しない。従って、CPU の割り込み禁止フラグや動作モードに関係なく、メモリ内容を取得することができる。また、測定対象プログラムの動作を止めることがないため、測定動作が与える影響（プローブ効果）を最小限に抑えることができる。

Physical Read は、1 回あたり 4〜2,048 バイトのデータを読み取ることができ、その位置に制限はない（ただし、アドレスの下位 2 ビットは常に 0 である）。このほか Physical Write 機能もあり、読み書きの違いのほかは Physical Read と同様に動作する。

なお、Physical Read/Write 機能は Open Host Controller Interface [6]（OHCI）規格に準拠のコントローラが搭載している[1]。現在流通している PC 用コントローラのほぼすべてが OHCI 準拠であり、デバイスの入手性は優れているといえる。

2.3 動作状態の取得

本研究では、Physical Read 機能を使い、カーネルのメモリ領域を読み取ることで OS の動作状態を取得する。

もし取得したいデータがメインメモリではなく CPU レジスタ等にある場合、被測定側でソフトウェアによる支援が必要となる。たとえば、レジスタ内の割り込み禁止フラグの場合、その切り換えルーチン中に、フラグ内容をメモリに書き出すコードを付加すれば、Physical Read によって値を取得することができる。

Physical Read を行う際、メモリキャッシュとの関係も考慮に入れる必要がある。しかしながら、Intel Pentium 系プロセッサでは、IEEE 1394 コントローラがメモリ内容を読み取るとき、前もってソフトウェアがメモリ領域をフラッシュないし無効化する必要がない。従って、ソフトウェアの処理がなくとも、Physical Read を使って最新のメモリ内容を得ることができる。逆に領域の無効化が必要なアーキテクチャでは、ソフトウェアによる対応が必要と考えられる。

1. USB コントローラにも同名の規格があるが、IEEE 1394 の OHCI とは異なる。

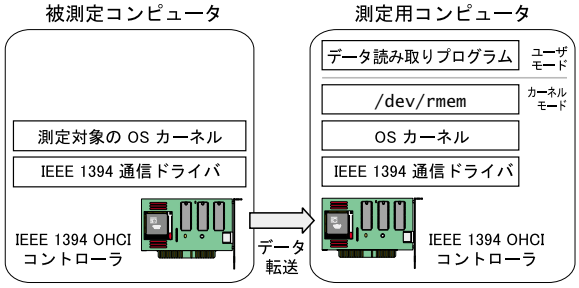

図 1　実装の全体構成

3 実装

実装には Intel Pentium III 600MHz、128MB SDRAM 搭載の PC/AT 互換機を 2 台用いた。それらに IEEE 1394 コントローラ（TI 社製 OHCI-Lynx 搭載、PCI ボード）を装着した。測定側では Linux 2.0.36、被測定側では Linux 2.2.14 を動作させた[2]。

測定用コンピュータでは、***FireCluster*** [5, 8] に用いられた IEEE 1394 ドライバを動作させ、その上にデバイスファイル `/dev/rmem` を導入した（**図 1**）。このファイルをとおして、ユーザプログラムは被測定コンピュータのメモリ内容を読み書きすることができる。

被測定コンピュータにもドライバを導入した。ただし、通信は Physical Read によって行われるため、デバイスの初期化後はドライバが処理を行うことはない。

3.1 通信性能

Physical Read を用いた通信の基本性能を計測した。具体的には、被測定コンピュータに 2KB の物理メモリ（ページアラインメント）を確保し、もう一方のコンピュータからは `/dev/rmem` を利用してその領域を繰り返し読み取った。1 回当たりの読み取りサイズを変化させ、サイズごとに 256MB 分のデータを読み取った。

その結果、読み取り間隔は最小 21 マイクロ秒（4 バイト時）、最大 85 マイクロ秒（2,048 バイト時）

2. IEEE 1394 ドライバの通信部分が Linux 2.0.36 にしか対応していないため、やむを得ずそのバージョンを使用した。なお、測定にあたって両コンピュータの OS、バージョンを一致させる必要はない。

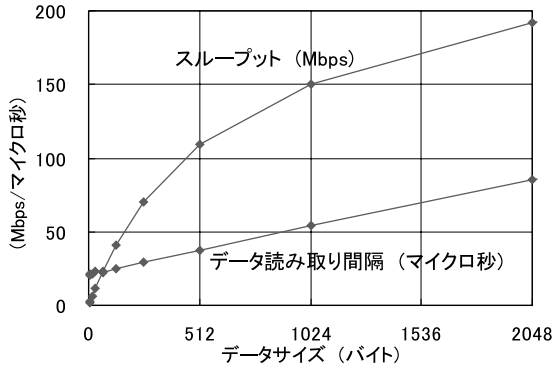

図 2　Physical Read の通信性能

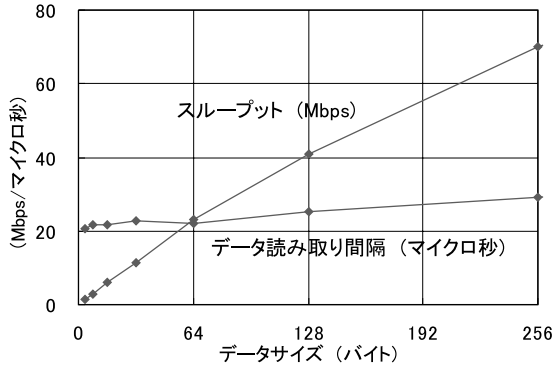

図 3　Physical Read の通信性能（拡大）

となった（**図 2**、**図 3**）。つまり、平均 21〜85 マイクロ秒間隔でカーネル内のメモリ内容をサンプリング可能である。

一方、読み取り間隔のばらつきについても調べた。データサイズが 8 バイトと 2,048 バイトのときについて 1 万回 Physical Read を繰り返し行い、1 回の転送にかかる時間の分布を取った。その結果、ばらつきは小さく、99% 以上が中央値付近に集中した（**表 1**、**図 4**）。この結果から、Physical Read の読み取り間隔は安定しているといえる。

通信のスループットは最大 192Mbps に達しており、測定データの転送用途としても Physical Read が有効といえる。

表 1　読み取り間隔の変化

サイズ	中央値	分散
8 バイト	23 マイクロ秒	1.9
2,048 バイト	91 マイクロ秒	8.2

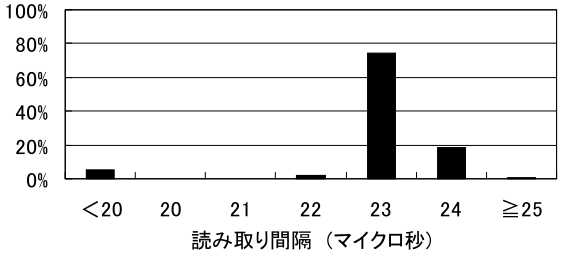

図 4　8 バイト Physical Read 時の読み取り間隔の分布

3.2　アクセス競合時の読み取り間隔

以上の結果は、被測定コンピュータがアイドル状態のときに得られたものである。しかし、Physical Read と CPU の両方からメモリアクセスが行われると、アクセス競合のために読み取り間隔が広がったり、不安定になる可能性がある。

これを確かめるため、同じメモリ領域に Physical Read とメモリ書き込みを同時に行っている状態で、読み取り間隔を計測した。その結果によると、メモリアクセスがある場合とない場合とで、平均・ばらつきとも有意な差は確認できなかった。従って、Physical Read はメモリアクセスの影響を受けにくいといえる。

一方、Physical Read を繰り返し行うと、被測定コンピュータの処理性能がわずかに低下することがある（**5.4.2 節**）。

なお、メモリ書き込みは memset 関数を使用し、ウェイトなしで繰り返し行った。ただし、先頭 8 バイトには被測定コンピュータの時刻（CPU の TSC: Time Stamp Counter[3]）を書き込み、それを読み取り間隔の計算に用いた。これによって、読み取り間隔のほかにも、最新のメモリ内容が Physical Read によって読み取れていることを確認できた。

4　プロセス、割り込み状態の測定

IEEE 1394 を用いた測定の例として、プロセスの動作状態、割り込みの状態等をリアルタイムに取得する実験を行った。

実験にあたり、測定対象の Linux カーネルに手を加え、新たに測定用データ `fp_struct`（**図 5**）を物理メモリ上に確保した。

動作中のプロセスに関する情報（`task_struct`

3. TSC は、CPU クロックに対応した 64 ビット幅のカウンタである。

```
struct firepeep {
  u32 id_mode;      PID と CPU 動作モード
  u32 addr;         task_struct のアドレス
  u32 eflags;       CPU フラグ
  u32 irq_on;       割り込み処理中なら 1,2,...
  u32 irq_count;    割り込みの累積発生回数
  u32 bh_on;        bottom half 中なら 1
} fp_struct;
```

図 5　測定用のデータ領域

スケジューリング部分（kernel/sched.c）

```
  kstat.context_swtch++;
  get_mmu_context(next);
  fp_struct.addr = (u32)next;
  fp_struct.id_mode =
    FP_SET_ID_MODE(next->pid, FP_KERNEL);
  switch_to(prev, next, prev);
```

割り込み許可マクロ（include/asm/system.h）

```
  #define __sti() do { __asm__ __volatile__ \
    ("sti": : :"memory"); \
    fp_struct.eflags = FP_EFLAGS_STI; } while(0)
```

図 6　カーネルの変更内容（一部）　下線部分を追加した。

構造体のアドレス）、割り込み許可フラグ等はメインメモリ上にないため、カーネル内の一部ルーチンにわずかな追加、変更を行い、fp_struct に最新の情報を反映するようにした（**図 6**）。測定用コンピュータでは、fp_struct のほか、各プロセスの task_struct を読み取り、プロセス名等を取得する。

実験では、被測定コンピュータで Linux カーネルのコンパイルを実行させ、そのときの動作状態を記録、プログラムによって図式化した[4]（**図 7**）。

これを見ると、プロセスの実行が順次 gcc（PID 12855）、make、gcc（PID 12856）に遷移していく様子がわかる。また、遷移はすべてカーネル動作中に起こっているのが確認できる。割り込み（IRQ）に着目すると、2 回目の発生直後、割り込み禁止状態（IF が 0）が 6 回の読み取り中に観測されている。データの読み取り間隔は平均 26 マイクロ秒（実験での実測値）なので、その時間はおよそ 156〜182 マイクロ秒となる。

以上のように、IEEE 1394 を使ってプロセスや割り込みの動作状態をリアルタイムに把握することができる。特に、従来の測定手法では把握しにくい割り込み許可フラグ（IF）の状態を観測することができた。

4. 図式化プログラムは、リアルタイムに動作状態を取得、描画することができる。

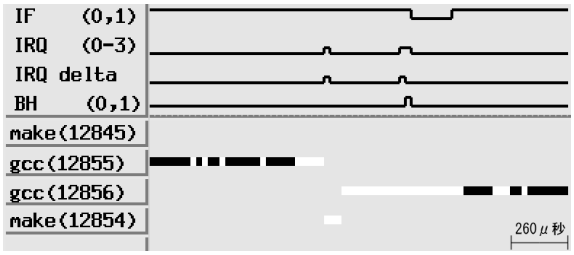

図 7　プロセス・割り込み状態の図式化　上段から、IF が割り込み許可フラグ、IRQ が割り込み処理中を示す値（ネストすると 2 以上）、IRQ delta が IRQ 発生回数、BH が bottom half 実行中のフラグである。下半分は動作中のプロセス名と PID で、黒線がユーザモード、白線がカーネルモードを示す。時間軸に沿って、左から右へ描画される。

```
77.969779946 tcp_v4_sendmsg
77.969783359 ip_queue_xmit
77.969784746 dev_queue_xmit
77.969786773 s004write (out)
77.969787519 s003read (in)
77.969788799 schedule pid:0
77.969799999 do_IRQ_handler (in) irq:5
```

図 8　本プロファイラの出力例　左からイベントの時刻（秒）、イベント名、イベント発生時の変数内容（オプション）を示す。

5　カーネルプロファイラ

前節の実験では、カーネル内に設けたデータ領域を「覗き見」することにより、動作状態の取得を行った。それとは別に、より汎用的な測定環境を実現するため、IEEE 1394 を利用したカーネルプロファイラ（以下、本プロファイラ）を開発した。

既存のカーネルプロファイラには、プロシージャや命令の実行回数、実行時間等の統計情報を出力するものが多い。一方、本プロファイラでは、それらの統計情報ではなく、どの処理がどのタイミングで実行されたのか、いわば処理の時系列をトレースし、出力するようにした（**図 8**）。こうすることで、統計情報からは得られにくい内容を知ることが可能となる。たとえば、メモリキャッシュ等の影響による実行時間の変化や、入出力処理とスケジューリング間の相互動作の様子が把握できる。

一方で、処理のトレース結果はサイズが大きくなりやすく、そのデータをどのようにして保存するか、リアルタイム性を失わずに対応することは難しい。そこで本プロファイラではデータの転送に IEEE 1394 を利用し、被測定コンピュータに与える影響を最小限に抑えながら、長時間の測定に対応することができる。

```
//@fm schedule pI 2 : *$s:"pid:%hu" = next->pid
```

図 9　ソフトウェアプローブの例　これはプロセス切り替えのコード部分に挿入されるもの。イベント名に schedule、ユーザデータに次回実行されるプロセスの PID（2 バイト）を指定している。

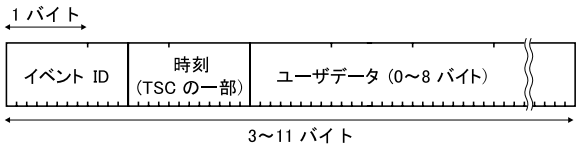

図 10　プローブが生成するイベントデータ

5.1　プロファイラの構成

本プロファイラは、IEEE 1394 のドライバ、/dev/rmem のほか、以下のプログラム等から構成される。

- 初期化等を行うカーネルソースパッチ
- ソフトウェアプローブの展開スクリプト
- /dev/rmem を使ったデータ読み取りプログラム
- データの整形出力プログラム

5.2　ソフトウェアプローブ

本プロファイラで測定する処理内容はユーザが指定する。具体的には、ユーザがカーネルソースコード中の任意の場所に、1 行のソフトウェアプローブ（**図 9**）を挿入する。本プロファイラのスクリプトはこの行を処理し、実際のプローブコード（C 言語、一部インラインアセンブル）に展開する。ユーザは展開後のファイルをコンパイルし、動作させて測定を行う。

実行時、プローブは測定「イベント」を発生させる。実際には、展開されたコードが処理の内容（イベント名）、時刻等を記録することであり、これをイベントの発生としている。

イベントの発生時、時刻だけでなく、任意の状態変数を収集できるとユーザにとって都合が良い。そこで、プローブ行には、イベント名のほか、ユーザが収集したいデータ（以下、ユーザデータ）を指定することができる。その際、データの型、出力形式、内容を表す式をコロン ':' 以降に記述する。このようにして、ユーザは printf と同じ感覚でソフトウェアプローブを埋め込んでいくことができる。

収集したデータは、プローブによって 3〜11 バイトのイベントデータ（**図 10**）にパックされ、メモリ上に格納される。プローブの実際の内容は、21〜26 個のアセンブリ命令 + ユーザデータ記録用の命令となる。

表 2　プロファイルデータ領域（2KB）

バイト数	内容
8	最新イベントの完全な時刻（TSC）
4	周回数と次回書き込み位置
4	（あき）
2,032	イベントデータ

5.3　測定データの転送

ソフトウェアプローブはマイクロ秒未満の間隔で実行されうるため、イベントデータは短時間のうちに膨大な量となる（1 秒あたり数 MB 以上）。この量に対し、リアルタイム性を失わずに対処することが必要である。そこで本プロファイラでは、イベントデータの転送に Physical Read を用いた。

具体的には、被測定コンピュータのメモリ上に 2KB（IEEE 1394 の最大ペイロード長に等しい）の領域を確保し（**表 2**）、そこに順次イベントデータを記録していく。領域の末尾まで到達したら、先頭に戻ってデータを上書きする。

測定用コンピュータでは、Physical Read を使ってこの領域を繰り返し読み取り、前回の読み取りに対する増分をディスクに保存していく。その際、領域先頭にある「最新イベントの完全な時刻」（TSC の値）をもとに、各イベントの時刻を復元する。同時に、イベントデータの取りこぼしがないかチェックし、もしあればユーザに警告を行う。

5.4　転送方法の検証

もしデータの保存・転送に IEEE 1394 を利用しない場合、ローカルディスクや Ethernet を使う手段が考えられる。しかし、デバイス操作は一般に低速なため、プローブの実行時間が大きくなり、リアルタイム性が損なわれる。また、転送デバイスから割り込みが発生するなど、本来とは異なった動作を観測するおそれがある。一方、本プロファイラの測定方式では、プローブ実行時にデバイス操作や割り込みが起こらないため、これら 2 つの問題が発生しない。

しかしながら、プローブの実行時間はゼロではなく、それが大きいとリアルタイム性に問題が生じる。また、測定用コンピュータから繰り返し Physical

表 3　Physical Read（P. R.）による被測定側の影響

getpid 1 万回にかかる時間		処理性能の低下
P. R. なし	P. R. あり	（左値の差より）
6.16 秒	6.27 秒	1.9%

Read が行われており、それによって被測定側の処理性能が低下するおそれがある。

以下では、その 2 点について検証を行った。

5.4.1　プローブのオーバヘッド

getpid システムコールを利用して、プローブの実行時間を調べた。具体的には、getpid の入口にプローブを挿入した場合としない場合とで、TSC を使って実行時間の計測を行い、その差を計算した。

その結果、プローブ 1 回あたりの実行時間は 100〜217 ナノ秒（600MHz で 60〜130 クロック）となった。測定の対象がプロセス切り換えや、ある程度まとまった処理の節目ならば、このオーバヘッドは許容できる範囲内といえる。逆にソースコード 1 行分といったごく短い時間内の測定には向かない。

なお、計測された実行時間に幅があるが、getpid の実行を単独で行うか、繰り返し行うかで大きな差が生じた。この原因はメモリキャッシュにあると考えられる。

5.4.2　Physical Read による処理性能の低下

3.2 節では、メモリアクセスと Physical Read が同時に行われるときの影響を調べた。その結果、Physical Read の実行性能は低下しないことがわかった。一方、メモリアクセスを行う側に処理性能の低下は見られないのか、本プロファイラでの様子を調べた。

具体的には、被測定コンピュータで getpid を 1 万回繰り返し、その実行時間を計測した。Physical Read を行っている場合とそうでない場合で時間に差が生じれば、その分、処理性能が低下したといえる。なお、getpid の入口には前節と同様プローブが挿入されている。

計測の結果、1.9% の処理性能低下が見られた（**表 3**）。これは許容できる範囲内といえる。

5.5　TCP/IP 処理の測定

本プロファイラを利用した測定の例として、TCP/IP でデータを送信する際にかかる時間を測定した。

測定にあたり、他コンピュータとの間でパケットの送受信を繰り返し行うユーザプログラムを用意した。ソフトウェアプローブは、write システムコール、カーネル内の主要な処理の開始位置に挿入した。

表 4　TCP パケット送信の測定結果

時刻（初回）	（1 万回目）	処理内容
0.00 μ秒	0.00 μ秒	write の開始
2.45	0.64	TCP 処理の開始
10.02	4.05	IP 処理の開始
11.51	5.44	ドライバ処理の開始
13.96	7.47	write の終了

結果を見ると、TCP のプロトコル処理が実行時間の大部分を占めていることがわかる（**表 4**）。また、通信処理のループの初回と 1 万回目について比べると、2 倍近くの差が生じている。プロファイラの出力によると、初回以降、徐々に実行時間が減っており、メモリキャッシュの影響が大きいと考えられる。

6　関連研究

MRSA [7] では、**4 節**と同様、カーネル内変数を繰り返しサンプリングして OS の動作を測定している。しかし、タイマ割り込みを利用しているため、割り込み禁止状態中の動作を観測しにくいという欠点がある。また、大量の測定データを効率良く扱うことが難しい。

カーネルを対象としたプロファイラには、IKD [3] と DCPI [2] がある。これらは、タイマ割り込みや、キャッシュミス回数に応じて発生する割り込みを利用し、プロシージャの実行クロック数、キャッシュミス頻度を測定している。どちらも、ある一定時間における平均値を出力するもので、処理のホットスポットを把握するのに有効といえる。逆に、処理の時系列を把握するのには適していない。

本研究のプロファイラと同様のイベントトレースを行うものに TinyTOPAZ [4] がある。これはデータの転送と保存に特殊なハードウェアを用いており、IEEE 1394 と比べて入手性が低く、一般的とはいえない。

7 まとめ

本研究では、OS の動作を把握する手段として IEEE 1394 を利用することを提案した。その利点は、プログラムの動作を一切止めることなく、また CPU の動作モードやフラグにかかわりなく動作状態を把握できることである。また、通信性能の評価の結果、動作状態のサンプリングは安定した間隔で行えることを確認できた。

さらに、IEEE 1394 を測定データの転送に利用した OS プロファイラを開発した。これはイベントトレースを行うもので、ユーザは printf と同様の感覚でカーネル内にプローブを埋め込み、測定を行うことができる。

本研究の提案する手法は、主にリアルタイム性が重要とされる処理に対して有効であるといえる。例えば、ネットワーク処理では、カーネル内の TCP/IP のパラメータや、通信ドライバの管理領域等を繰り返し読み取ることで、動作状態の細かな変化を把握することができると見込まれる。その際、Physical Read によって、リアルタイム性を損なわずに測定できることは重要といえる。

プロファイラの改良点として、プローブのオーバヘッドの削減、プローブの実行時挿入等が考えられる。また、得られたデータの分析手法についても検討を行う予定である。

謝辞

IEEE 1394 の通信ドライバの最新版を提供していただいた兵頭和樹氏、ならびに測定結果の図式化プログラムを提供していただいた佐藤喬氏に深く感謝いたします。

参考文献

[1] Don Anderson, “FireWire System Architecture: IEEE 1394a,” 2nd ed., MindShare, Inc., Addison-Wesley, August 1999.

[2] Jennifer M. Anderson *et al.*, “Continuous Profiling: Where Have All the Cycles Gone?,” ACM Trans. Computer Systems, pp. 357–390, November 1997.

[3] Andrea Arcangeli, “IKD patch,” `http://www.kernel.org/pub/linux/kernel/people/andrea/ikd/` (as of February 1, 2001).

[4] Takashi Horikawa, “TinyTOPAZ: A Hybrid Event Tracer For UNIX Servers,” Proc. 1999 Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS), pp. 203–210, July 1999.

[5] 兵頭 和樹, 中山 泰一, “IEEE 1394 を用いた PC クラスタシステム—通信機構の設計と評価”, 情報処理学会論文誌: ハイパフォーマンスコンピューティングシステム, pp. 39–47, 2000 年 11 月.

[6] The Promoters of The 1394 Open HCI, “1394 Open Host Controller Interface,” Release 1.00, October 1997.

[7] 多田 好克, “MRSA と et による OS の振舞いの定量的な測定”, 情報処理学会第 42 回プログラミング・シンポジウム報告集, pp. 71–82, 2001 年 1 月.

[8] 山之内 暢彦, 兵頭 和樹, 南 将朝, 中山 泰一, “IEEE 1394 による PC クラスタシステムの設計”, 情報処理学会第 58 回全国大会講演論文集, 3F–01, 1999 年 3 月.