システムソフトウェアと
オペレーティング・システム 84－22
（2000．5．25）

OSの性能評価を可能とする命令トレーサの開発

森本 洋行† 池谷 敬之†† 毛利 公一††† 吉澤 康文†††

†東京農工大学大学院工学研究科
††（株）日本トータルシステム
†††東京農工大学工学部

マルチメディアアプリケーションやネットワークサーバは，オペレーティングシステムへの依存度が高く，その性能をオペレーティングシステムに左右される場合が多い．このような場合には，アプリケーションプログラムとオペレーティングシステムを同一環境において，同時に性能測定を行う必要がある．本稿では，アプリケーションとオペレーティングシステムの性能を総合的，定量的に測定，評価を行うための命令トレーサ『鶴』と解析ツール群『鶴 Tools』の設計と実装について述べる．『鶴』は Intel x86 アーキテクチャ上で動作する Linux Ver2.0.34 に実装されている．『鶴』を用いることにより，AP と OS の性能を定量的に測定することが可能となる．

Instruction Tracer for the Performance Evaluation of Operating System

Hiroyuki MORIMOTO† Hiroyuki IKEYA††
Kouichi MOURI††† Yasufumi YOSHIZAWA†††

†Graduate School of Engineering, Tokyo University of Agriculture and Technology
††Japan Total System Corporation
†††Faculty of Engineering, Tokyo University of Agriculture and Technology

It is necessary to evaluate the total system performance of application program and operating system, because, application programs depend on operating system In order to evaluate the total performance, instruction tracer had been developed. In this paper, we describe that concept of design, implementation of instruction tracer "TURU", and analyzing tools "TURU Tools." TURU enables to evaluate the total performance of application program and operating system. TURU is implemented on Linux Ver2.0.34 running on Intel x86 architecture.

1 はじめに

マルチメディアアプリケーションや，ネットワークサーバなどの性能は，オペレーティングシステム（以下，OS と記す）への依存度が高く，その性能に左右されることが多い．スケジューリング方式や，メモリ管理方式などがアプリケーション（以下，AP と記す）の性能に影響を与える．OS の提供する資源管理方式と AP が利用する資源管理方式の整合性がとれている場合に最も高い性能が得られる．しかし，汎用 OS では多くの AP に適応できるような方式が選択される．マルチメディア AP およびネットワークサーバは，汎用 OS を利用しながら，専用のサーバマシンとして設置される．これらのサーバマシンに適した資源管理方式を提供可能な OS を作成することにより，現在より高性能なサーバマシン OS として利用することが可能である．

このような OS を開発するためは，現状での性能評価を行い，その問題点を洗い出すことが必要である．しかし，このような場合，AP または OS 単体での性能評価値は意味を持たない．これらの性能評価を行うためには，AP と OS を同一環境において，同時に評価しなければならない．その上で，性能低下の原因が AP によるものか，OS によるものかを考察し，原因を特定する必要がある．しかし，OS と AP の性能評価指標を同一の環境で同時に取得する手段は提供されていない．

以上の問題点を解決するために，我々は，OS と AP の性能を総合的，定量的に測定，評価を行うことを目的とした，命令トレーサ『鶴』と解析ツール群『鶴 Tools』を開発した．『鶴』は，AP とそこから呼び出される OS の処理を機械語レベルで記録する．この記録データを『鶴 Tools』を用いて解析することにより，対象プログラムのダイナミックステップ数，参照メモリアドレス，割込みの発生などを調べることが可能であり，これらは性能評価指標の 1 つとなる．これにより，OS と AP の性能を定量的に評価し，オーバヘッドの大きな部分を見つけ出すことが可能となる．

以下，本稿では 2 章で『鶴』の概要，3 章で『鶴』の構成，4 章で Linux に対する実装を述べる．また，5 章で評価，6 章で関連研究を述べ，7 章でまとめを述べる．

2 『鶴』の概要

2.1 目的

『鶴』は，AP と OS の依存関係を明確にし，その性能を定量的に評価することを目的とする．したがって，AP プロセスとそこから呼び出される OS の処理を記録することが必要となる．『鶴』が目的とする性能評価の例を次に示す．

(1) 実行命令列とダイナミックステップ数

プログラムのダイナミックステップ数を計測することにより，様々な評価を行うことができる．例えば，OS の処理では各システムコールを実行する場合のダイナミックステップ数を計測することにより，その応答時間を概算することが可能である．応答時間はリアルタイム性を保証するための評価指標として重要である．ネットワークサーバにおいても，1 つのクライアント要求を処理する場合のダイナミックステップ数は重要である．この値を元に，その計算機で処理受付可能な最大クライアント数を試算することが可能である．また，1 つのクライアント要求を処理する場合の各部分のダイナミックステップ数を計測し，命令の種類を調べることにより，オーバヘッドの大きな部分の特定や改善が可能となる．

対象とする CPU アーキテクチャにおける命令の使用頻度分布や，平均命令長を求めることも性能評価や，その改善に有効である．このデータを元に命令の追加や削減など，CPU の持つ命令セットの変更を行うことができる．

(2) 参照したメモリアドレス

ダイナミックステップ数と同様に，メモリ参照も，プログラムの性能を評価する指標となる．メモリ参照は，CPU 内部の計算処理と比較して低速であり，計算機全体の処理能力低下の原因となる．この問題を解決するために，OS ではセグメントやページングを用いてメモリ参照の局所化を試みている．また，ハードウエアでは，キャッシュを用意することにより，データ参照の高速化を行っている．しかし，マルチメディア AP では，特にデータ転送量が多く，メモリ参照の局所化を行うことは難しい．『鶴』を用いて，マルチメディア AP やネットワークサーバなどのメモリ参照の局所性や，そのアクセスパターンを調べることにより，効率的なメモリ配置や，メモリの先読みアルゴリズムを求めることができる．キャッシュの利用効率が向上することは，計算機全体の処理効率の向上につながる．

(3) 割込み

割込みを捕らえることができるのは，OS が実際に動作しているときだけである．このような動的測定対象に対して『鶴』は有効である．例えば，リアルタイムシステムでの応答性を調べるために，割込み禁止区間のダイナミックステップ数を把握することは重要である．

(4) システムレジスタの変更

システムレジスタには，現在の特権レベルや割込みの許可などが保存されている．また，ページフォルトが発生した際のリニアアドレスなどの CPU フォルト情報もシステムレジスタに記録される．これらの値は，計算機の状態を表す情報となる．

2.2 記録データ

『鶴』は，プログラムの実行を機械語レベルで命令トレースする．『鶴』が記録するデータは次のものが挙げられる．

(1) 機械語命令列

プログラムの実行を機械語レベルでトレースし，記録する．このデータを利用して，プログラムのダイナミックステップ数や命令セットを求めることができる．

(2) 参照したメモリアドレス

参照したメモリアドレスは，命令列のリニアアドレスと，そのメモリ参照オペランドのリニアアドレスを記録する．

(3) 割込み

発生した割込みの種類と，それに付随する情報（エラーコードなど）を記録する．

(4) システムレジスタの変更

変更されたシステムレジスタと，その変更後の値を記録する．これにより，条件分岐の成否や，特権レベルの変更，アドレス空間の変更がわかる．

2.3 『鶴』の起動と終了

『鶴』の起動と終了は，対象プログラムのソースコードにシステムコールを埋め込む．起動時にカーネル，ライブラリ，AP のシンボルテーブルを記録する．シンボルテーブルには次のものが含まれる．

(1) オフセットアドレス
(2) シンボル名
(3) シンボル名の有効範囲

シンボルテーブルを記録することにより，プログラムの実行を C 言語の関数を単位として追跡することが可能となる．

3 『鶴』の構成

『鶴』の命令トレース対象プログラムの機械語命令が実行されるたびに，『鶴』へ制御が移行する．『鶴』は，次に実行される命令列を解析し，記録データとして出力する．この機能は，シングルステップ実行実現部，トレースデータ生成部，データ出力部の 3 つの部分から構成される．また，『鶴』の出力したデータを解析するためのツール群が AP として実現されている．それぞ

れの詳細を次に述べる.『鶴』の基本動作を図1に示す.

3.1 シングルステップ実行実現部

シングルステップ実行実現部では,1つの命令が実行されるごとに,『鶴』に制御を移行させる機構を提供する.

3.2 トレースデータ生成部

対象プログラムの命令トレースと,割込みの発生,システムレジスタの変更を監視する機構を提供する.

(1) 命令デコード部

次に実行される命令のデコードを行い,命令レコードとして記録する.命令長,オペランドの個数などを判定する.また,参照したリニアアドレスを取得する.

(2) 割込みイベント処理部

割込みが発生した際に,割込みイベントレコードを記録する.

(3) システムレジスタ監視部

システムレジスタの変更を監視する.システムレジスタに変更があった場合には,システムレジスタレコードを記録する.

3.3 データ出力部

『鶴』の生成したデータを記録する機構を提供する.

(1) 静的メモリエリアへの保存

静的メモリ書出しを利用する場合には,スワップアウト対象外のメモリの一部を『鶴』のデータ出力領域として確保する.『鶴』は,この領域にトレースデータを出力する.『鶴』終了後に,ユーザプロセスによって,メモリからハードディスク上のファイルシステムへデータの保存を行う.

(2) ファイルシステムへの保存

命令トレース対象プログラムが大きく,そのトレースデータが膨大になる場合がある.このような場合には,『鶴』起動中に逐次ファイルシステムへデータ保存を行えるよ

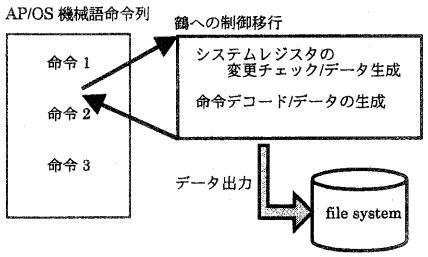

図1 『鶴』の基本動作

うな機構を提供している.

3.4 トレースデータ解析ツール群

トレースデータを解析するためのツール群『鶴 Tools』を作成した.これらは,APとして実現される.したがって,ユーザの目的に応じて自由に拡張したり,新規作成することが容易にできる.試作的に作成したツールを次に示す.

(1) 各レコードデータ表示ツール

各レコードの値を表示する.このツールを基本として,他のツールを作成する.

(2) 関数遷移表示ツール

『鶴』起動時に取得したシンボルテーブルと各命令のリニアアドレスのマッチングを取ることにより,C言語の関数を単位とした,関数遷移を調べ表示する.

『鶴』ではOSの処理もそのトレース対象として含まれる.APからのシステムコール発行によるOSへの処理依頼や,その際に発生した割込みやスケジューリング処理もグラフィカルに表示することが可能である.

(3) メモリアクセス分布表示ツール

各命令のリニアアドレスを元に,メモリアクセス分布を調べ表示する.

4 実装

『鶴』は,Intel x86アーキテクチャ上で動作するLinux(Ver.2.0.34)に実装されている.『鶴』のLinuxへ実装した場合の構成を図2に示す.以下,Linuxに実装する際の詳細について述べる.

4.1 処理の流れ

命令トレーサ『鶴』の核となるシングルステップ実行実現部とトレースデータ生成部のコードを図3および図4に示す．図4は割込みハンドラの一例として，システムコールが発行された際の割込みハンドラである．図4はトレースデータ生成部である．

4.2 シングルステップ実行の実現

命令のシングルステップ実行を実現する方法として，次の2つが挙げられる．

(1) シングルステップ例外

これは，Intel x86 アーキテクチャが提供するデバッグ支援機能の1つである．これにより，1つの命令実行が終了するたびにデバッグ例外が発生する．

(2) 機械語コードの動的書換え

メモリ上に展開された実行コードに対して命令の置換えを行い．1命令ごとに『鶴』へ，制御を移行させる．Jump 命令および call 命令など，プログラムカウンタを書換える命令を検出して，その飛び先番地の命令に対して命令の置換えをする必要がある．

『鶴』では実現の容易さと，この機能が Linux で利用されていない点より，シングルステップ例外を利用した．『鶴』では割込みディスクリプタテーブル（以下，IDT と記す）の置換えを行い，シングルステップ実行による命令トレースを行う．また，Linux が持つ割込みハンドラは『鶴』の命令トレース対象である．したがって，『鶴』が置き換えた IDT からシングルステップ例外をセットした状態で，Linux の割込みハンドラに処理が移行する．

『鶴』を利用するには，他のプログラムによる IDT の書換え，置換えは許されない．『鶴』では，IDT を置き換える命令を監視する．その命令が実行される場合には『鶴』は強制終了される．

```
ENTRY(tracer_system_call)
   SAVE_REGS
   SET_TRECE_FLAG
   call STORE_INT_EVENT
   RESTORE_REGS
   jmp SYMBOL_NAME(system_call)
```

図3 『鶴』の割込みハンドラの例

```
void single_step(struct registers *regs)
{
   BYTE_T buff[];
   disable_interrupt();
   if(check_system_regs(regs,buff)
                        == CHENGED)
      store_data(buff);
   if (decode_instruction(regs,buff)
                        == PAGE_FAULT)
      return ;
   else
      store_data(buff)
   enable_interrupt();
   return;
}
```

図4『鶴』の処理内容

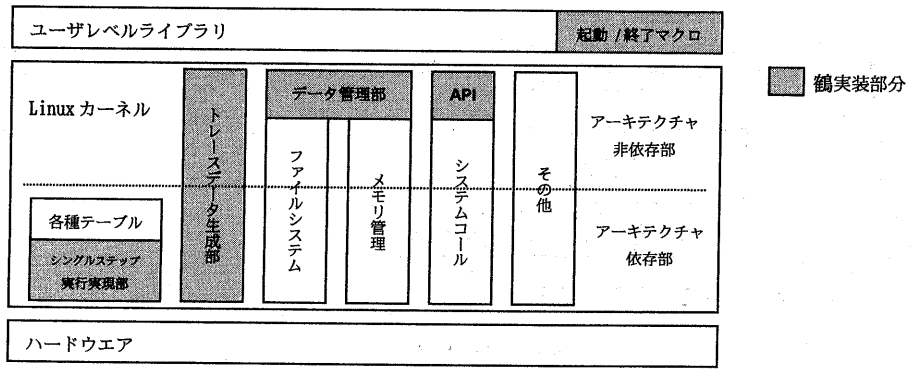

図2 命令トレーサ『鶴』の構成

4.3 命令レコード生成部

命令レコード生成部は，『鶴』の核となる部分である．この部分の効率によって『鶴』のオーバヘッドが決定される．次に命令レコード生成部の機能を3つに分けて説明する．

(1) 命令分類による命令長判定

Intel x86 アーキテクチャは命令体系とし命令長が可変長である．命令レコードの生成を高速化するために，命令を命令長とオペランド指定形式によって分類する．

(2) ページフォルトの予測

『鶴』が命令列を取得する際にページフォルトが発生する可能性がある．これは，『鶴』が命令トレース対象となるプロセスよりも先に命令列を取得するためである．『鶴』は命令列取得中にページテーブルの参照を行い，ページが実メモリにロードされていない場合にはその命令のデコードを中止し，トレース対象プロセスに制御を戻す．対象プログラムの実行が再開されると，ページフォルトが発生し，OS の提供するページフォルトハンドラが呼び出される．このページハンドラによって，ページがスワップインされ，命令が実行される．次の命令に対して『鶴』が呼び出された際に，再度前命令列の取得と記録レコードの生成を行う．

(3) レコードの生成

取得した命令列を記録レコードのフォーマットにしたがって整列し出力する．命令レコードフォーマットを図5に示す．

4.4 トレースデータ保存方法

トレースデータ保存のインタフェースとして，次の2種類のインタフェースを提供する．

(1) 静的メモリエリアへのデータ書出し

カーネル起動時，物理メモリ上にトレースデータ保存領域を確保する．この領域はスワップアウトの対象外である．この方式の長所は，データの保存コストが低い点と，データ出力に伴ってハードウエアからの割込みが発生しない点である．欠点は，データ領域が小さいことである．トレースデータが膨大になり，メモリオーバフローが発生した場合には，『鶴』は強制終了される．『鶴』が強制終了された場合には，その旨を表示し，強制終了時点までのデータを静的メモリエリアに残す．この問題は大容量メモリを搭載することにより解決可能である．

静的メモリエリアに蓄積されたデータは，『鶴』の終了後に，カーネル空間のメモリをユーザ空間に転送するシステムコールを利用して，ユーザプロセスによってファイルシステムに保存される．

(2) ファイルシステムへの逐次書き出し

『鶴』による命令トレース実行中に，逐次ファイルシステムへデータを書き出す．入出力効率を考慮して，64kB の書込みキャッシュを持つ．この方式は，トレースデータが膨大になることが予測される場合に有効である．

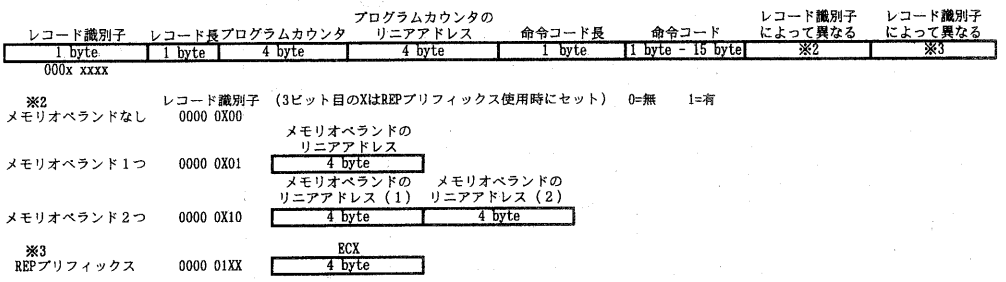

図5 命令レコードフォーマット

4.5 実装規模

『鶴』の実装規模を表1に示すとおりである．使用言語はC言語および，アセンブリ言語である．

表1 実装規模

追加個所	追加行数
環境設定部	700
命令デコードおよびデータ生成部	1,600
(命令分類表を含む)	500
メモリ管理部	600
ファイルシステム出力部	
合計	3,400

5 評価

5.1 使用例

命令トレーサ『鶴』と解析ツール群『鶴 Tools』の使用例として，Linuxのシステムコール getpid をトレースした．システムコールはログラム実行時に動的にロードリンクされる．したがって，同一のプロセスにおいて，1回目のシステムコール実行と2回目のシステムコール実行では，そのダイナミックステップ数が異なる．getpid を例としてダイナミックステップ数の違いを測定した．結果を表2に示す．また，2回目の getpid の関数遷移を『鶴 Tools』を用いて表示した．結果を図6に示す．getpid はその処理が簡単であり，OS による実装差が少ないため例として用いた．

表2 getpid のダイナミックステップ詳細

	1回目		2回目以降	
	実行命令数	全体比	実行命令数	全体比
ユーザプログラム	7	0.9%	2	3.1%
DLL	685	10.6%	82	12.4%
カーネル	82	88.5%	12	84.5%
合計	774	100%	97	100%

5.2 『鶴』のオーバヘッド

『鶴』が命令トレースを行ない，記録データを出力することによるオーバヘッドを測定した．まず，システムコール getpid を実行した際のクロック数を『鶴』起動時と非起動時で計測した．計測結果を表3に示す．システムコールだけを計測したため，アドレス空間の切替えが発生し，システムレジスタレコードが出力されるために，『鶴』によるオーバヘッドが約1430倍と大きい．一般的なプロセスを測定した場合には，全体のオーバヘッドは320倍程度である．また，各部の占める割合を図7に示す．

表3 鶴のオーバヘッド

	クロック数
鶴起動時	143,300
非起動時	100

```
Program Start!
<main>:user                  1steps
  ->:library
    :library                 5steps
      ->:kernel
        = total 36steps
      <-
    :kernel                  8steps
      ->:kernel
        = total 49steps
      <-
  <-
:user                        = total 12steps
-----------------------------------------
Total dynamic instruction = 97steps
user level =  2[ 3.1%]
library    = 12[12.4%]
kern level = 82[84.5%]
```

図6 getpid の関数遷移

5.3 生成されるデータ量

『鶴』によって生成される記録データ量は，次に示すとおりである．

T = Sym + N×inst

T：合計

Sym：シンボルテーブルサイズ(Min. 100kB)

N：定数（平均 17.2 Byte/inst.）

inst：命令数

6 関連研究

OS の定量的測定は，古くから研究されている．UNIX システムコールの実行時間から，その性能を定量的に測る試みがなされている[2]．また，メモリの効率的利用を目的とした，メモリ使用率分析なども行われている[3]．また，UNIX にはシステムコール ptrace なども実装されている．しかし，ptace はプロセスやスレッドを対象としており，OS の処理を命令トレースすることはできない．

7 おわりに

本稿では，Linux カーネルの一部として動作する命令トレーサ『鶴』の設計と実装について述べた．

本研究により，OS と AP の性能評価指標を同一の環境で同時に取得する手段として，命令トレーサ『鶴』と解析ツール群『鶴 Tools』を提供することができた．『鶴』は，AP の動作とそこから呼び出されるカーネルの動作を機械語レベルで記録する．この記録データを解析することにより，対象プログラムのダイナミックステップ数，参照メモリアドレス，割込みの発生などを調べることが可能であり，これらは性能評価指標の 1 つとなる．『鶴』と『鶴 Tools』を用いることにより， OS の性能を評価し，オーバヘッドの大きな部分を見つけ出すことが可能となった．

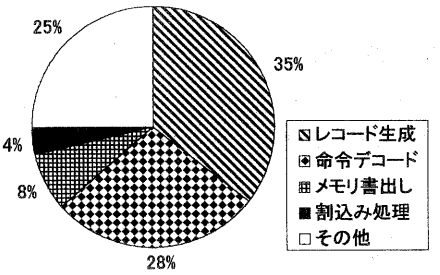

図 7 オーバヘッドの内訳

参考文献

[1] 池谷，吉澤：性能評価のための命令トレーサの開発，情報処理学会第 58 回全国大会公演論文集(1)，pp.123-124，1999.

[2] 中村，野地：UNIX システムコールの性能評価，情報処理学会研究報告，pp.41-48，1991.

[3] 杉村：RTS のメモリ使用の分散的分析手法と実施例，情報処理学会研究報，pp.81-88，1995.