デバッガのためのプログラム疑似逆実行方式

寺田 実†

本論文は，再現性のある逐次プログラムを対象として，デバッグに有用な逆実行の方法を提案するものである．本方式の特徴は，コンパイラのプロファイリング機構を流用することによりCPUやOSに依存しないポータブルな実装を行っていること，任意の位置から関数呼び出し単位での逆実行ができること，逆実行先として現在関数の先頭やその親といったプログラムの構造上自然な場所への移動を可能にするアルゴリズムを示したことの3点である．さらにオーバヘッドの計測を行って，実用的な性能が得られていることも確認した．

Pseudo Reverse Execution for Debugging

Minoru Terada†

In this paper we present a method for pseudo reverse execution of sequential, deterministic programs. Our method has these three features: (1) By exploiting the profiling facility of compilers, we achieved portable method for reverse execution; (2) The control can be returned to the beginning of any function calls; (3) As for the target of reverse execution, we can designate important point such as the entry of the current function, or its caller. We also measured the overhead of our method, and certified its efficiency.

1. はじめに

デバッギングとは，結果から原因を求めるという本質的に時間軸に逆行する過程である．プログラム上の誤り（バグ）を原因として，誤った結果の出力，異常終了，無限ループなどの兆候（結果）が得られ，その兆候をもとに真の原因を推測していくのが通常のデバッグ手法である．仮にプログラムが逆に実行できるのであれば，デバッギングの手法としてきわめて有効であることは容易に想像できる．

本論文では，実行に再現性のある逐次的プログラムを対象として，デバッグのための逆実行方式を提案する．関数呼び出しのそれぞれに順次番号（タイムスタンプ）を付与し，対象プログラムの先頭からの再実行によって任意の時点への「逆戻り」を行うのである．これによって，異常の発覚地点から出発が可能になる．

先頭からの再実行による逆実行というアイデア自身は新規なものではないが，本研究は以下の点に特徴を持つ：

- タイムスタンプの付与方法—CPU，OSなどに依存せず，既存のツールで実現可能．
- 逆実行の単位—関数呼び出し単位での任意の点への逆実行可能．
- 逆実行先の指定方法—プログラムの構造上自然な場所への移動が可能．

本論文の構成は，まず2章で逆実行に関するこれまでの研究を概観し，本研究の方法と比較する．続いて，3章ではタイムスタンプ付与の方法を説明する．4章では，単純なタイムスタンプによる逆実行における問題点を指摘し，本論文で述べるアルゴリズムの必要性を示す．5章ではそれをうけて，プログラムの構造上自然な位置への移動を可能にするためのアルゴリズムについて記述する．6章で以上の仕組みをまとめる．7章では，実装したシステムについて述べ，オーバヘッドの計測を行う．8章は結論である．

2. デバッギングと逆実行

一般にプログラムの実行は不可逆である．手続き型言語における代入は，不可逆性の最大の要因であるが，分岐命令による制御の合流やブロック構造から出る際の局所変数の消滅なども同様の問題をもたらす．

一方，デバッギングとは，結果から原因を求めるという本質的に時間軸に逆行する過程である．手作業によるデバッグ作業を考察すると，ブレークポイントによって実行を中断して異常の有無を調べる操作を繰り

† 東京大学大学院工学系研究科
Faculty of Engineering, University of Tokyo

返し，誤りの存在範囲を狭めていく．その際，エラーの発覚地点から実行を遡る形でブレークポイントを設定していくことはよく見られる方法であり，これはつまり手作業による逆実行と考えることができる．

計算機による逆実行支援は，以下のように大別される．

2.1 処理系への組み込み

典型的にはインタプリタによる実行系に見られるが，プログラムが不可逆な処理を行う際に，それを回復するための情報を内部的に保存するという方式である．ZStep 95[1]では，Scheme インタプリタに修正を行い，プログラムの動作に加えてウィンドウ操作も含めて GUI プログラムの逆実行を可能にしている．

この方式の最大の問題点は特殊化された処理系を必要とすることで，他言語への適用，他システムへのポータビリティ，処理系のアップグレードなどに困難を生じる．

2.2 状態の保存と回復

プログラムの内部状態とは基本的にはメモリ内容であるから，それをファイルなどに保存し，あとで回復すれば保存時点に戻ったことになる．

この方式では，メモリ内容という低いレベルでの保存を行っているため，言語の種類や処理系の実行形態（コンパイラ/インタプリタ）には依存しにくく，ポータビリティにも優れている．しかし，状態保存ファイルのサイズの問題と，保存間隔の問題がある．

IGOR[2]では，前者についてはメモリ管理システムの協力を得て変更のあったメモリページのみを保存することとし，後者については，保存時点の中間への移動は，まず目標点の直前の保存時点でのメモリを回復し，そこから専用インタプリタによって順方向実行を行うとしている．

2.3 ステップ実行コマンド系列の再実行

デバッガ下での実行を対象とする．ユーザが発行したステップ実行コマンドの履歴を保存しておき，プログラムの開始点からそれを必要数だけ再実行することで，ステップ実行の停止点系列の任意の点に戻ることが可能になる[3]．

再実行による逆実行は，一般にプログラムの再現性を必要とする．しかし，これは手作業によるデバッグにおいても必要な性質であり，それほど重大な制限ではないと考える．

対話的デバッグでは有用な方式であるが，過去に一時停止した点にしか戻れず，逆実行というよりは実行コマンドの undo と考えた方が正しい．特に，異常の発覚点からスタートすることができない（あらかじめその点に至るステップ実行が必要）のは大きな問題点である．

2.4 タイムスタンプの付与

プログラム実行におけるある種のイベントにタイムスタンプを与えることができれば，それを指定して先頭から再実行することで任意の場所に制御を移すことができる．プログラムに再現性を要求するのは上の方法と同じである．

これが本研究で対象とする逆実行の基本方式で，関数の呼び出しを単位として，タイムスタンプを付与することとする．

3. タイムスタンプによる実行制御

3.1 関数呼び出しへのタイムスタンプの付与

関数呼び出しにタイムスタンプを付与するには，いくつか方法がある．

まず，コンパイラを修正して，関数の入口にタイムスタンプ更新のコードを挿入することが考えられる．しかし，これはシステムのポータビリティに大きな問題点が残り，現実的な方法とはいえない．

次に，デバッガ下での実行を対象として，すべての関数入口にブレークポイントを設定し，そこで停止するたびにデバッガ側で管理するタイムスタンプを更新する方式がある．これは，対象プログラムと処理系にまったく変更が不要であるという利点があるものの，オーバヘッドが大きく，やはり現実的ではない．

そこで本研究で着目したのが，コンパイラが提供するプログラムのプロファイリング機構の流用である．いくつかのコンパイラでは，コンパイル時にオプションを指定することにより，プログラムにプロファイリングのためのコードを付加する．gcc の場合は，`-pg` オプションを指定すると関数の入口部分に `mcount` という関数への呼び出しが挿入される．この `mcount` は，本来は関数の呼び出し回数の計測用で，その定義はシステムのライブラリに含まれている．

この `mcount` として，呼び出されるたびにカウンタをインクリメントする関数を当方で定義し，それを `-pg` オプションによる呼び出し先としてリンクする．これはいわばシステム提供の `mcount` を「横取り」したかたちになっており，この方法によって，すべての関数呼び出しにタイムスタンプを与えることがポータブルに実現できることになった．

3.2 タイムスタンプを指定しての停止

前述の方法により，すべての関数呼び出しにタイムスタンプを与えることができた．次は，プログラムを先頭から実行し，指定した任意のタイムスタンプの呼

び出しでプログラムを停止させる方法である．停止後の制御はデバッガに戻すことになるため，直接的な停止の方法はブレークポイントによることになる．

1つの方法は前節で述べた mcount 関数に条件つきブレークポイントを置き，タイムスタンプと目標値の一致を調べるものである．これは結局，関数呼び出しごとにデバッガの介入を招く．

もう1つは，mcount の中でプログラム自身の手で目標値との比較を行う方法である．目標値と一致した場合には別の関数を呼び出すこととし，その関数に（無条件）ブレークポイントを設定すればよい．この方法では，目標値までの実行過程にはデバッガの介入がなく，オーバヘッドをおさえることができる．なお，プログラムのアドレス空間に目標値を外部から設定することになるが，これはデバッガコマンドにより可能である．

3.3 タイムスタンプを利用した実行制御

本章で述べてきた方法で関数呼び出しに付与したタイムスタンプを用いて，プログラムの実行制御が可能になる．

まず，特定の関数呼び出しのタイムスタンプを何らかの方法で記録しておけば，その時点へ戻ることが可能である．

プログラム中のある位置で停止しているとして，その時点でのタイムスタンプを N とすると，目標値を $N+1$ として実行継続することで，次の関数呼び出しの先頭まで進むことができる．

一方，目標値を N として先頭から再実行すれば，現在までの最後に行われた関数呼び出しの先頭に戻ることができる．

以上のように，当初の目標であった逆実行が実現されたように見える．しかし，ユーザの立場からは，これには問題がある．それについては次章で述べる．

4. タイムスタンプによる逆実行の問題点

プログラムの実行にともなう関数の呼び出し関係は，節点を個々の関数呼び出しとして，木構造で表現できる．以下では，例として家系図に即したプログラム（**図 1**）を用いる．その実行木を**図 2** に，時系列としてのプログラムの実行を**図 3** に示す．実行は「自分」関数の内部で停止しているとする．図 3 で斜線を施した○印は関数の入口点であり，前節の mcount の呼び出しのタイミングを表している．

プログラムの実行は，実行木の深さ優先トラバースに対応する．深さ優先トラバースにおける節点の訪問のタイミングについては，pre-order，post-order の 2

```
長兄 (){}
次兄 (){}
長男 (){}
次男 (){}
自分 (){
  長男 (); 次男 (); ... <現在位置> ...
}
父 (){
  長兄 (); 次兄 (); 自分 ();
}
```

図 1 プログラム例
Fig. 1 A sample program.

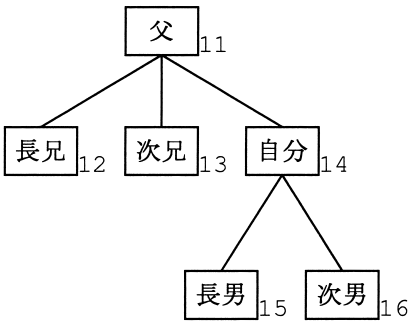

図 2 図 1 の実行木
Fig. 2 The call tree for the program.

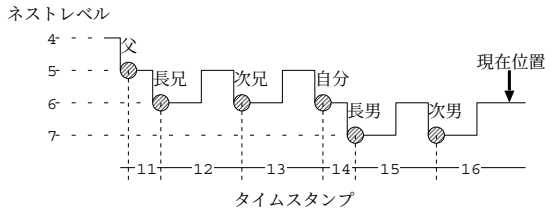

図 3 図 1 の実行
Fig. 3 The execution diagram for the program.

つがあるが，関数の入口点をもって「訪問」とすると，pre-order となる．前章で付与したタイムスタンプは，まさにこの順になるのである（図 2 では，タイムスタンプを節点の右側に示す）．

プログラムが順方向に実行中であれば，この順序はそれほどの違和感がない．しかし，逆方向に実行する際にはこれが大きな障壁となる．現在の停止点から過去（左）に向かって最初に出合う関数（の入口点）を考える．図 3 ではそれは自分の次男となるが，次男に子孫がいた場合にはさらに再帰的に末子となるものがそれに該当する．これはプログラム構成上は非常に離れた地点である．

したがって，逆実行の際には，単純な「直前点」ではなく，ユーザの直観的な理解が容易な停止点を選ぶ必要がある．そのためにはプログラムの静的な構造が重要であると考えた．つまり,「自分」関数の内部から逆実行するのであれば，停止点としては望ましいのは以下であろう：

- 「自分」関数の入口点
- 「父」関数の入口点

実行制御における静的構造の重要性を補強する事実として，逆実行ではなく順実行に関するものであるが，デバッガのコマンド体系をあげることができる．たとえば gdb では，ステップ実行コマンドとして，単純に次に実行される文まで（したがって，現在位置が関数呼び出しであれば，その関数の先頭まで）進むもの（step）と，現在位置が関数呼び出しであれば，その終了まで（すなわち現在位置のプログラム上の次の文まで）進むもの（next）の 2 種類があり，利用者は必要に応じて使い分ける．

さて，実際に「自分」や「父」の先頭を目標として逆実行を行うためには，それらのタイムスタンプの目標値を獲得しなければならない．しかし，これまで述べた方法（mcount の利用）に沿ってそれらを獲得する方法は自明ではない．次章で，そのためのアルゴリズムを述べる．

5. 目標値の獲得アルゴリズム

「自分」や「父」の先頭への逆実行に必要となる，それらのタイムスタンプの獲得のためのアルゴリズムを述べる．まず，利用可能な情報やオーバヘッドなどに関する制約条件を最初に示し，続いてアルゴリズムの記述を行う．

5.1 制約条件

前述の mcount を横取りする方法で，利用可能な情報には 2 つの種類がある．プログラム自身で獲得できる情報と，デバッガが介入することで獲得できる情報である．

デバッガはプログラムの実行時環境を参照しているだけであるから，これら 2 つの間には本質的な違いはない．しかし，ポータビリティを考えると大きな違いがある．たとえば，スタックフレーム中に保存されるリンク情報や帰り番地などは，プログラムからポータブルに知る方法はない．これに対して，デバッガはそもそも個々の処理系に合わせて構成されているため，統一的なコマンドによりそれらの情報にアクセスが可能である．以下ではこれら 2 つの種類についてまとめる．

5.1.1 プログラム自身で得られる情報

前述のとおりスタックフレームの形態がポータブルでないため，プログラム自身ではそこに含まれる情報は利用できない．利用できる情報の 1 つは，mcount に渡される引数である．これは元来の目的である関数呼び出し回数の計測のために関数ごとに静的に割り当てられたワードの番地である．

もう 1 つは，フレームそのもののスタック中での位置が利用できる．mcount の中で局所変数を割り当てると，それはスタック上に場所を持つため，その番地を知ることで mcount を呼び出したフレームの位置が（ある偏差を持って）得られる．以下ではこの値を $FP(activation)$ と記すことにする．

FP の値については，次の関係だけが一般的に成立する☆．

$$FP(\text{父}) > FP(\text{子})$$

引数をスタックに積んでから関数呼び出しを行うコード生成の場合，引数個数によって子のフレームの位置が変化するため，とくに兄弟の場合であってもその FP は等しくならない．したがって，mcount の FP だけからでは，前回のmcount の呼び出しが父のものであるのか兄のものであるのかは区別ができないことになる．

5.1.2 デバッガの介入で得られる情報

デバッガは対象プログラムの実行状態について詳細な情報を提供する．ここで利用するものとしては，以下がある：

- 関数呼び出しのネストのレベル
- 現在の関数に至る直系の先祖の関数名（スタックトレース）

ここで注意が必要なのは，先祖のタイムスタンプは入手できない点である．タイムスタンプ自身は単一の変数であり，それが常時更新されているためである．

5.1.3 オーバヘッドに関する要件

本方式はデバッガでの利用を念頭においているため，オーバヘッドを現実的な範囲におさえることは重要である．ここでは以下の 2 点を考える．

- デバッガ介入による時間的オーバヘッドをおさえるため，mcount で毎回介入することはしない．
- メモリに保持すべき情報を抑制するため，たとえば関数呼び出しの全履歴を保持することはしない．

5.2 アルゴリズム（1）— mcount での処理

「自分」や「父」のタイムスタンプを求める処理は，

☆ これはスタックが高位番地から 0 番地に向かって伸びる実装を想定した場合である．

```
タイムスタンプ ++
fp <- 現在のフレーム位置（FP）
frame_position[0] .. frame_position[level_index] の中で
   frame_position[i] > fp >= frame_position[i+1]
  となる i を探索 (0 <= i < level_index)
(a) そのような i が存在
  level_index <- i+1
  frame_position[level_index] <- fp
  frame_timestamp[level_index] <- 現在のタイムスタンプ
(b) そのような i が存在しない --- つまり (frame_position[level_index] > fp)
  level_index ++
  frame_position[level_index] <- fp
  frame_timestamp[level_index] <- 現在のタイムスタンプ
```

図 4 アルゴリズム（1）—mcount でのスタックフレームのシミュレーション
Fig. 4 Stack frame simulation in mcount.

```
C <- 現在のタイムスタンプ
LEVEL_INDEX <- level_index
FRAME_TIMESTAMP[0..LEVEL_INDEX] <- frame_timestamp[0..level_index]
L <- 現在のネストレベル （デバッガの where コマンドで取得）

プログラムを強制終了
プログラムを先頭から再実行
FRAME_TIMESTAMP[I] で停止させ (I = 0 .. LEVEL_INDEX)
        LL <- 現在のネストレベル （where コマンドで取得）
        (LL == L) ならば
                P <- FRAME_TIMESTAMP[I]

ここで最終的に得られた P が求めるタイムスタンプ
```

図 5 アルゴリズム（2）—「自分」のタイムスタンプを求める処理
Fig. 5 Algorithm to find the timestamp of the current function.

(1) データ収集を目的としてすべての関数呼び出しについて mcount 内でプログラム自身が実行する処理
(2) 一時停止した地点から実際に目標値を求める処理

の 2 段階に分かれる．これらを順に説明する．

mcount 内では，スタックフレームの成長をシミュレートし，それぞれに対するタイムスタンプを保存しておく．そのために以下のデータ領域を用意し，**図 4** に示す処理を mcount で行う．

- スタック frame_position[N]
  トップレベルから自分に至るフレームのスタック内の位置 FP を保存する．値は単調に減少することになる．5.1.1 項の末尾で注意したとおり，直系の先祖以外のフレームも含まれる可能性がある．領域のサイズ N は，プログラム実行における最大ネスト数程度の大きさが必要である☆．
- スタック frame_timestamp[N]
  上の配列の要素であるフレームのタイムスタンプを保存する．
- 変数 level_index
  上の 2 つに対するスタックポインタである．初期値は -1 とする．

5.3 アルゴリズム (2) — 一時停止点での処理

タイムスタンプを求める基本的な方針は，その時点

☆ 父のフレームと自分のフレームとの間に，兄たちのフレームがすべて混入した場合には必要となる N は増加する．その場合でも，スタックの最大使用量（これは最大ネスト数の定数倍と見積もることが可能）を超えることはない．

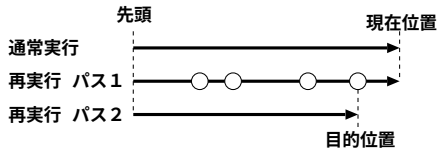

図 6 逆実行の手順
Fig. 6 Steps for reverse execution.

での frame_timestamp[] の中から,「自分」や「父」のタイムスタンプを選び出せばよいわけだが，それには十分な情報がない．そこでプログラムを先頭から再実行し，候補となるフレーム（つまり frame_timestamp[] の要素）ごとに実行を停止してデバッガ介入による検査を行い,「自分」や「父」であるかどうかを確認することにする．

ここで,「自分」であることの確認は,

「自分」のネストレベルと等しいレベルを持つ,

時間的に最後のフレーム

という基準で行う（「父」の場合には,「自分」より 1 つ少ないレベルを持つ最後のフレームとなる）．

アルゴリズムの詳細は，**図 5** に示す．なお，大文字で示した変数は，プログラム中ではなくデバッガ側で保持する変数である．

6. 逆実行方式のまとめ

本論文で提案する逆実行の手順をまとめる（**図 6**）．

(1) プログラムのコンパイル
  (a) 個々のファイルを -pg オプションを指定してコンパイルする．これによって，すべての関数の先頭に mcount への呼び出しが挿入される．
  (b) あらかじめ用意した mcount をリンクして実行ファイルとする．この mcount には，図 4 のアルゴリズムを実装してある．

(2) 順実行
  プログラムを先頭から通常実行する．その際，mcount により，図 4 のアルゴリズムも実行される（5.2 節で述べた方法）．プログラムを適当な場所で停止させ，次の逆実行段階へ移る．

(3) 逆実行
  (a) 再実行パス 1
    プログラムを先頭から再実行する．その際，図 5 のアルゴリズムにしたがって候補地点（図 6 の○印）でデバッガを介入させ，最終的に目的位置のタイムスタンプを獲得する（5.3 節で述べた方法）．

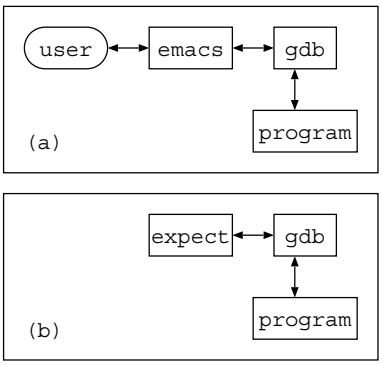

図 7 システムのプロセス構成
Fig. 7 Process organization for debugging.

  (b) 再実行パス 2
    プログラムを先頭から再実行し，目的位置のタイムスタンプで停止させる（3.2 節で述べた方法）．

7. 実装と計測

7.1 実　　装

以上述べてきた逆実行の仕組みを，ユーザのデバッグ作業に活用するため，**図 7** に示す構成のシステムを構築した．構成（a）は人間が対話的に利用するためのもの，構成（b）は後述する性能計測やバッチ的な情報取得のためのものである．

（a）では，mygdb と名付けた Emacs のための gdb のインタフェースに付加するかたちで実装し，逆実行のために gdb を操作する部分（図 5 のアルゴリズム）を Emacs Lisp で記述した．キー割当てとして,「T」を現在実行中の関数の先頭への移動,「U」を現在関数の親の先頭への移動としている．

（b）における expect とは，人間用に作成された対話型プログラムをプログラムから制御するためのツールであり，最初の実装においては Tcl ベースのもの[4]を利用していた．現在の版では，Perl 上に実装されたものを利用している．

7.2 オーバヘッドの計測

ツールとしての実用性を実証するため，本方式の実行時オーバヘッドを計測した．以下の 2 点に起因するオーバヘッドに注目する．

- 関数呼び出しごとの mcount の呼び出し
- デバッガ介入による多数プロセスのインタラクション

対象プログラムとして，関数呼び出しが中心となる hanoi(20) を選んだ（**図 8**）．計算機は PC 互換機，CPU は Pentium 90 MHz，Memory 40 MB，OS は

```
void hanoi(int n, int f, int t, int v)
{
  if(n > 0) hanoi(n-1, f, v, t);
  // move(n, f, t);
  if(n > 0) hanoi(n-1, v, t, f);
}

main()
{
  hanoi(20, 100, 200, 300);
}
```

図 8 計測用プログラム：hanoi
Fig. 8 The benchmark program: hanoi.

Linux 2.2.6 である．結果を**表 1** に示す．

7.2.1 mcount の横取りによるオーバヘッド

関数呼び出しごとに mcount を呼び出すことによるオーバヘッドを求めるため，プログラムを最後まで連続実行させ，経過時間を計測した．計測は以下の項目である：

(1) 通常のコンパイルによる実行
(2) mcount を横取りした実行
(3) 図 7 (b) のプロセス構成で (2) を実行—これは expect や gdb の起動/終了のオーバヘッドを見るために測定した

7.2.2 タイムスタンプによる停止のためのオーバヘッド

関数呼び出しにタイムスタンプを与え，その特定の値で停止させるためのオーバヘッドを求めた．hanoi の 2097150 回目（ほぼ最後）の呼び出しで停止させている．

(4) hanoi にブレークポイントを設定し，毎回デバッガと expect が介入（これについては，10000 回での所要時間から外挿した）．
(5) デバッガの条件つきブレークポイントを利用し，expect の介入を避ける．
(6) デバッガの回数つきブレークポイントを利用し，expect の介入を避ける．
(7) ユーザプログラムが自分でタイムスタンプを比較する方式（本論文で提案する方式）．

7.2.3 現在関数の先頭への逆実行

(8) 本論文で述べたアルゴリズムを用いて，hanoi の 2097150 回目の呼び出しで停止させたのち，その関数呼び出しの先頭への逆実行を行わせた．図 6 に示したとおり，これには合計 3 回の実行が含まれ，とくに逆実行目的位置がプログラムの最後に近いため，最も時間のかかる逆実行となっている．

表 1 オーバヘッドの計測結果
Table 1 Measurement of the overhead.

項目	経過時間（秒）
(1) 通常実行	0.62
(2) mcount 横取り	1.65
(3) 図 7 (b)	3.84
(4) ブレークポイント（毎回介入）	∼ 46000.00
(5) 条件つきブレークポイント	1960.48
(6) 回数つきブレークポイント	1396.12
(7) ユーザプログラムが自分で比較	4.03
(8) 逆実行	11.56

7.2.4 計測結果の考察

計測結果からは以下の考察ができる：

- mcount の横取り (2) は，通常の実行 (1) に対して 2.6 倍の時間を要している．
- タイムスタンプの付与と目標値での停止については，本論文で提案した方式 (7) によれば通常実行 (1) の 6.5 倍で可能であり，他の方法 (4，5，6) よりずっと高速である．
- 逆実行については，合計 3 回の実行を行っているが，逆実行そのものはそのうちの後の 2 回で，その所要時間は (8) – (7) で 7 秒程度となる．
  本方式では，逆実行を再実行により実現しているため，逆実行の所要時間は現在位置までの実行時間に比例することになる．そこで，その比率をオーバヘッドと考えると，通常実行 (1) と比べて約 12 倍のオーバヘッドである．対象としたプログラム（hanoi）がほとんど関数呼び出しだけで構成されていることを考えると，このオーバヘッドは最悪値と考えてよく，デバッグ作業への有効性を考えると十分実用的な範囲であるといえる．

8. おわりに

関数呼び出しにタイムスタンプを与えるポータブルな方法を述べ，さらにそれを利用してプログラム構成上自然な場所への逆実行を行うためのアルゴリズムを記述した．

そのオーバヘッドを計測した結果，最悪の状態でも逆実行の所要時間が通常の実行の 12 倍程度であり，デバッグツールとして十分実用になる性能を持っていることを示した．

最後に，本論文で提案したmcount にブレークポイントを設定する方法により，従来のステップ実行では対応が難しかったコールバック関数スタイルのプログラミングに対しても容易にステップ実行が可能になることを指摘しておく．

ウィンドウプログラミングに典型的な，イベントによるコールバックのプログラミングスタイルにおいては，メインループ自身はライブラリとしてあらかじめ用意されており，デバッグ用オプションを含まない．したがって，通常のデバッガによりメインループの呼び出しの段階でステップ実行を行うと，メインループの終了まで制御が戻らず，途中に発生するコールバックで停止させることができない．そのためには，個別のコールバック関数にブレークポイントを置くことが必要となる．

これに対して，本論文で述べたように，ユーザ側で記述するコールバック関数群を適切なオプションを用いてコンパイルしておけば，コールバックが発生した時点でプログラムを停止させることができる．

参 考 文 献

1) Lieberman, H. and Fry, C.: ZStep 95: A Reversible, Animated Source Code Stepper, *Software Visualization*, MIT Press (1997).
2) Feldman, S. and Brown, C.: IGOR: A System for Program Debugging via Reversible Execution, *ACM Sigplan Notices*, Vol.24, No.1, pp.112–123 (1989).
3) Rosenberg, J.: *How Debuggers Work*, Wiley (1997). 吉川（訳）：デバッガの理論と実装，アスキー出版局 (1998).
4) Libes, D.: expect: Curing Those Uncontrollable Fits of Interactivity, *Proc. Summer 1990 USENIX Conference* (1990).

(平成 11 年 9 月 30 日受付)
(平成 12 年 5 月 11 日採録)

寺田　実（正会員）

1959 年生．1981 年東京大学工学部計数工学科卒業．1983 年同大学院工学系研究科情報工学修士課程修了．同大学計数工学科助手，電気通信大学電子情報学科助手を経て，1991 年東京大学工学部機械情報工学科講師，1992 年同助教授．工学博士．ソフトウェア科学会，ACM 各会員．