関数単位疑似逆実行の高速化

丸山一貴[†] 寺田実[††]

デバッガを用いて関数呼び出し単位の疑似逆実行を実現する方法の１つとして，プロファイル用の関数を置き換えることにより，Cプログラムにおける関数呼び出しの入口を捕捉する方法が提案されている．しかし，この方法では関数からリターンするところを捕捉できないので，逆実行時の停止目標点を決定するために，(1) 現在の実行点までスタックをシミュレートしながら先頭から再実行し，制御点の移動目標を決め（scan pass），(2) 実際に制御点をそこに移すために再度先頭から実行する（re-execute pass）ことになっており，本来不要なパス (1) が存在する．本論文では，コンパイラのアセンブリ言語出力を修正して関数からリターンするところをも捕捉し，スタックのシミュレーションを通常実行時に正確に行うことによって，従来手法では２パスを要していた逆実行時の処理を，１パスで行う手法を提案する．また，通常実行時のオーバヘッドと逆実行時のオーバヘッドを，経過時間の計測によって従来手法と比較する．

One-pass Pseudo Reverse Execution of C Programs

Kazutaka Maruyama[†] and Minoru Terada[††]

We propose a method of speeding up pseudo reverse execution of C programs by locating the target function call efficiently. In the former method, the profiler function, inserted by the compiler at the prologue of each function is overridden so that the program records necessary informations about stack frame for later use. When reverse execution is instructed, we (1) restart the program from the beginning, traces stack and decides the target point where the control should reach (scan pass), and then (2) restart it again and stop it at that point (re-execute pass). In our new method, we modified an assembly language output of C compiler so that we can catch the exit of function calls. So we can simulate stack to know depth of the stack at run time and decide the target point without “scan pass”. Measurement of the time of reverse execution and the overhead at normal execution and at reverse execution is also included.

1. はじめに

プログラミングにおいてバグはつきものであり，多くのバグはプログラム実行時に発見される．このような「ランタイムに発覚するバグ」を調べるための支援ツールとして，デバッガがある．デバッガを用いることで，プログラマは変数の値や制御の移動など，実行中のプログラムの状態を調べることができる．デバッガを用いた主なデバッグ手法は，およそ以下のようである：

(1) 誤りの原因と思われる場所の直前にブレークポイントを設定し，プログラムをそこまで実行する．
(2) プログラムを１行ずつ動かす．
(3) 停止した場所での変数の値を調べる．

このように，プログラムを任意の場所で停止させ，その時点での実行状態を検査することが基本となるため，デバッガがデバッグ対象プログラム（以下，デバッギという）の実行を制御できることはきわめて重要である[1)]．通常のデバッガはデバッギをその実行順どおりにしか動かすことができないので，現在位置の直前の情報を得るためにさえ，プログラマはデバッギの実行を停止させて，先頭から再実行させなくてはならない．

ところが，「プログラムの現在位置」に再び戻ってくるのは容易ではない．たとえば，単純にソースコードの現在行にブレークポイントを設定することを考える．そこがループの内部であったり，再帰的に呼び出される関数の内部であったりすれば，先頭から再実行してそのブレークポイントで停止したとしても「かつ

† 東京大学大学院工学系研究科情報工学専攻
Department of Information Engineering, Graduate School of Engineering, The University of Tokyo
†† 東京大学大学院工学系研究科
Graduate School of Engineering, The University of Tokyo

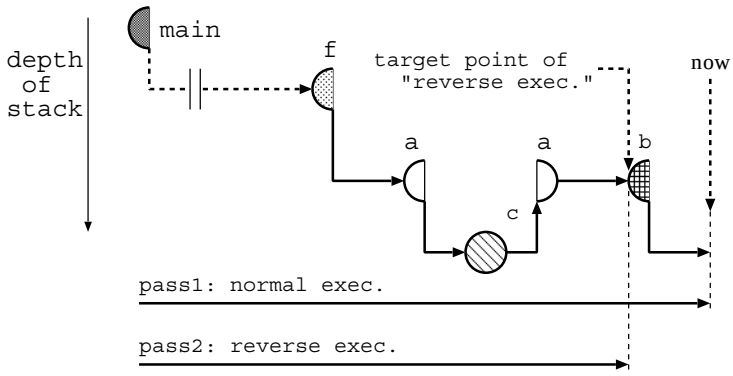

図 1 関数単位疑似逆実行
Fig. 1 Pseudo reverse execution stepping every function calls.

ての現在位置」ではないことがありうる．また，現在位置を同定するためには，種々の変数の値を比較する必要がある．プログラミング中のプログラマがこれを手動で何度も行うことは困難であり，デバッガによるサポートがあればプログラマの負担を軽減することができる．

これまでにも逆実行を実現する研究は行われてきたが，プログラミング言語の持つ機能のサブセットにしか対応していない場合や，実行速度が非常に遅く実用に耐えないなどの問題点があり，実際に利用されるには至らなかった[2]．

そこで，実行状態を逆向きにたどることによって完全な逆実行をするのではなく，人間が手動で行う逆実行プロセスを既存のデバッガを用いて自動化する方式，すなわち疑似逆実行方式が考案された[3]．しかし，文献 3) で提案された実現手法（以下，従来手法）では逆実行時に 2 パスの再実行を要するという性能上の問題点がある（詳細は 3.2 節で述べる）．本論文では，この逆実行時の再実行を 1 パスにして高速化するための実現手法を述べ，経過時間の計測結果を示す．

以下，2 章では関数単位疑似逆実行について説明し，そこで必要となる「タイムスタンプ」を導入する．3 章で新手法のアルゴリズムと従来手法との比較を述べ，4 章でその実装について説明する．5 章では計測の結果と高速化の要因を示し，6 章でまとめを述べ，7 章で今後の課題を提示する．

2. 関数単位疑似逆実行

この章では，まず関数単位疑似逆実行とは何かを述べ，その実現に必要な「タイムスタンプ」を導入する．

2.1 疑似逆実行

人間がデバッガを用いてデバッグしていると，デバッガの操作ミスによって，あるいは勘違いによって，本来見るべき「バグが発生する地点」を通り過ぎてしまうことがある．こうしたとき，人間は

(1) 実行中のデバッギを停止させ，
(2) 先頭から再度実行し直し，
(3) 先ほど通り過ぎてしまった場所まで，ブレークポイントやステップ実行によって制御を進める，

という逆実行手順を手動で行う．

逆実行に関する先行研究[2),4)]は，Lisp の S 式や C のステートメントといった単位でプログラムの実行状態を保持しておき，逆実行時にはそれを用いて状態を回復する，という考え方に基づいている．

これに対して前述の人間による逆実行手順は，プログラムを先頭から実行し直すことで疑似的に逆実行するものである．これを疑似逆実行と呼び，関数呼び出しを逆実行の単位とするものを関数単位疑似逆実行と呼ぶ．デバッギが再現性のあるプログラムであれば，この処理を自動化できる．

図 1 に，関数 b の内部を実行中に逆実行を指示する例を示す．プログラマがデバッギを実行して，now で示した現在位置まで進んできた（pass1）とする．ここで関数単位疑似逆実行を指示すると，逆実行の移動目標点を「関数 b の入口」と定め，プログラムを先頭から再実行してその目標点に制御を移す（pass2）．

2.2 タイムスタンプの導入

疑似逆実行では実行途中の状態を保持する必要はないが,「関数 b の入口」のように逆実行の戻り場所を特定する必要がある．

関数呼び出しを単位として逆実行を行うため，プログラム実行中のすべての関数呼び出しについて，1 から順に番号を割り振ることを考え，この番号をタイムスタンプと呼ぶことにする．タイムスタンプと関数呼び出しは 1 対 1 に対応するので，現在実行中の関数呼び出しの入口に戻るためには，そのタイムスタンプが

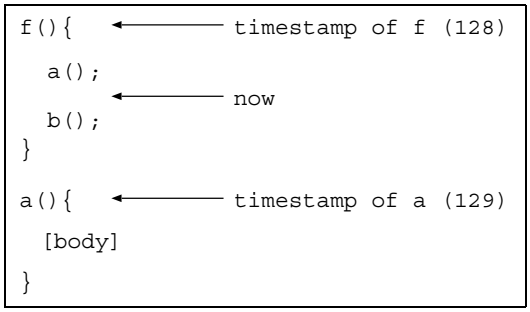

図 2 タイムスタンプの更新
Fig. 2 Changing timestamp.

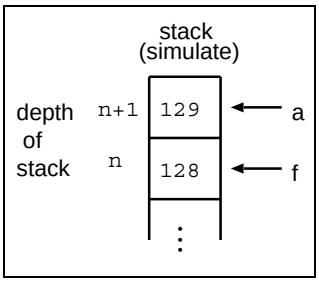

図 3 シミュレートするスタック
Fig. 3 Simulated stack.

分かればよいことになる．しかし，これはそれほど容易ではない．たとえば**図 2** に示す状況では，now で示した時点では関数 f を実行中であるのに，そのときのタイムスタンプ値はその直前の関数 a の呼び出しによって更新されていて，現在のタイムスタンプ値は利用することができない．

2.3 スタックのシミュレート

タイムスタンプは関数呼び出しに固有の情報であるから，スタックフレーム内に持てばよい．しかし，スタックフレームの形を変更してしまうのはポータビリティを損なうことにつながる．そこで，デバッギの通常実行中にスタックをシミュレートして，現在のフレームでのタイムスタンプ値をスタックの構造に保存しておくという方法をとる．

図 2 の状況を考えると，now の時点でのシミュレート中のスタックは**図 3** のようになる．

関数 f の呼び出しがスタックの n 段目で，そのタイムスタンプ値が 128 だったとすると，関数 a の呼び出しは 129 になり，その値がスタックの $n+1$ 段目に保存される．関数 a の実行が終わり，図 2 中の now で示した時点に制御が到達したところで逆実行を指示する場合は，現在のスタック深さである n 段目の値を調べると 128 であることが分かるので，現在実行中の関数（この場合は関数 f）の入口に戻るには，タイムスタンプ値が 128 の関数呼び出しで停止すればよいことが分かる．

本論文では，このスタックのシミュレートの仕方について，従来手法よりも高速な手法を提案する．

3. 新 手 法

本章では，スタックをシミュレートするための機構について，本論文で提案する新手法のアルゴリズムを説明し，従来手法との差異を述べる．

3.1 新手法のアルゴリズム

新手法では前述のシミュレートを実現するために，シミュレート用のスタックを操作するための関数（以下，フック関数と呼ぶ）を，関数呼び出しの入口と出口で呼び出すようにコンパイラの出力を修正する．関数呼び出しの入口でフックとして呼ばれる関数を mcount1，関数呼び出しの出口でフックとして呼ばれる関数を mcount2 と呼ぶことにすると，ソースコードはいわば**図 4** のように変換されるのと同じである．

この mcount1 では，関数呼び出しの入口であるから「現在のスタックの深さ」と「タイムスタンプ」をインクリメントし，新しい「タイムスタンプ」をシミュレート中のスタックの「現在のスタックの深さ」のところに保存する．mcount2 では，関数呼び出しの出口であるから「現在のスタックの深さ」をデクリメントする．

3.2 従来手法との比較

従来手法は，フック関数を入口だけ，つまり mcount1 だけにするという点が異なる．

OS のライブラリには，プロファイル用関数という関数呼び出し回数を測定するための関数が付属している☆．GCC などのコンパイラにはプロファイル用オプション☆☆が用意されており，これを指定するとすべての関数呼び出しの入口で必ずプロファイル用関数（以下，mcount）が呼び出されるようにコンパイルされる．つまり，3.1 節で述べた mcount1 の呼び出しを挿入する部分は，コンパイラが標準的に持っていることになる．

従来手法では，ここで OS のライブラリに含まれる mcount ではなく，自作の mcount を利用することにより，

- 専用の処理系が不要
- mcount を記述するだけでよく，記述量が少ない
- アーキテクチャ，OS，コンパイラに依存しない

☆ SunOS や Linux では mcount という名前のプロファイル用関数が OS のライブラリに含まれる．
☆☆ GCC では -pg オプションを用いる．

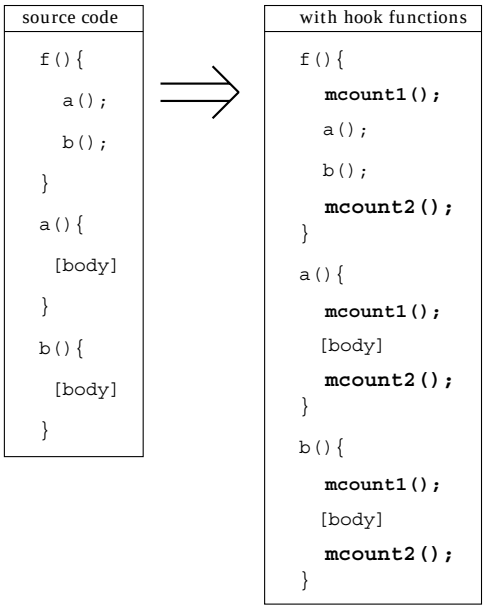

図 4 関数呼び出しの入口・出口の捕捉
Fig. 4 Insertion of hook functions.

という特長を実現している．

その一方で，mcount2 にあたる仕組みが存在しないために，関数からリターンしてスタックが浅くなることが分からず，スタックの深さを正確に認識することができない．そこで逆実行の際に，以下の 2 回の再実行パスが必要となる：

(1) デバッギを先頭から再実行し，デバッガの持つ，スタックを検査するコマンド☆を用いて，スタックの深さを正確にシミュレートしながら，目標点のタイムスタンプを調べる（scan pass）．
(2) scan pass では目標点を通り過ぎてしまうので，目標点に実際に制御を移すために，もう一度先頭から再実行をする（re-execute pass）．

3.1 節で示した手法を用いればこの scan pass が不要となり，1 パスで関数単位疑似逆実行を実現できる．

4. 新手法の実装

本章では，3 章で述べた新手法の実装について述べる．

4.1 フック関数呼び出しの挿入

mcount1 の呼び出しを挿入する機構は，コンパイラが持つプロファイル用オプションの仕組みを流用できる．一方，mcount2 の呼び出しを挿入する際には注意が必要になる．

関数呼び出しの出口ではその関数の返り値がレジスタに保存されているので，mcount2 の処理を行う前にこれを退避し，処理が終わったところでレジスタに回復しておかなければならない．この保存の仕方には 3 つの方法がありうる：

ソースコードレベル ソースコードを変換して return や関数定義ブロックが終わる場所の直前に，返り値の保存・退避と mcount2 呼び出しのコードを挿入する．

中間コードレベル コンパイラ内部に変更を加えて，関数のエピローグを処理する部分で上記の処理を行うようにする．

アセンブリコードレベル 返り値が保存されていると考えられるすべてのレジスタの内容をスタックに退避して，mcount2 を呼び出し，戻ってきたところでスタックからレジスタに復元する．

第 1 の方法は，返り値の型によって保存する変数の型を変えなければならず，ソースコードの解析が必要になり，あまり容易ではない．また，デバッグ時に見えるソースコードがプログラマが書いたものと異なるため，戸惑う要因となりうる．第 2 の方法は，ユーザが意識せずに利用することができ，かつポータブルな実装が可能なので，最も有力な方法である．第 3 の方法は，アーキテクチャとコンパイラに依存した変更になるため多数の実装を用意しなくてはならないが，個々の実装はこれらの方法の中では比較的容易である．本論文では実際に逆実行に要する時間が短縮されることを示すため，実装が容易な第 3 の方法を用いた．

実装は Intel Architecture，Linux，gcc-2.8.1 の環境に対して行った．mcount2 の呼び出しを挿入する部分は，コンパイラのアセンブリコード出力へのフィルタとして実装した．このフィルタを GCC のコンパイラドライバの中に組み込み，ある特別なオプションが指定されると，GCC のコンパイル手順の中でこのフィルタを自動的に通すようになっている．

mcount2 呼び出しのコードを挿入した例を**図 5** に示す．図中の矢印で示した部分が，実装したフィルタによって挿入された部分である．GCC が返り値を保存する可能性のあるレジスタのうち，mcount2 が利用する eax，edx，ecx の各レジスタを pushl でスタックに保存しておき，mcount2 の呼び出しから戻った後，popl でこれらを復元している☆☆．

☆ GDB であれば where コマンドがこれにあたる．

☆☆ mcount2 内部では浮動小数点は扱わないため，FPU レジスタの保存・回復は行っていない．

```
.L1:
    pushl %eax
    pushl %edx
    pushl %ecx      saving the return value,
    call mcount2      calling mcount2
    popl %ecx         and restoring it.
    popl %edx
    popl %eax
    movl -24(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

図 5 Intel Architecture の mcount2 呼び出し
Fig. 5 Calling mcount2 for Intel Architecture.

```
#define MAX_DEPTH 256
int stack_ts[MAX_DEPTH];
int depth = -1;
int timestamp = -1;
int ref = -1;

void mcount1(void){
  depth++;
  timestamp++;
  stack_ts[depth] = timestamp;
}

void mcount2(void){
  depth--;
}
```

図 6 フック関数の実装
Fig. 6 Implementation of hook functions.

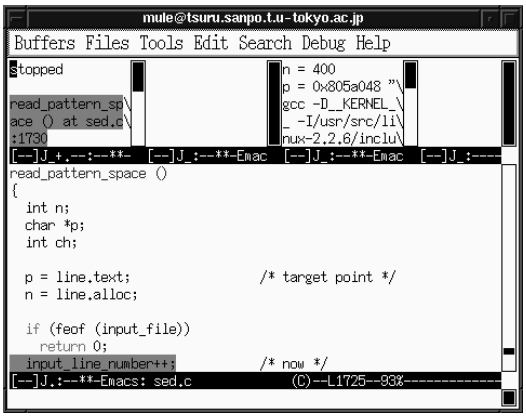

図 7 mygdb での利用例
Fig. 7 Example of using this method in mygdb.

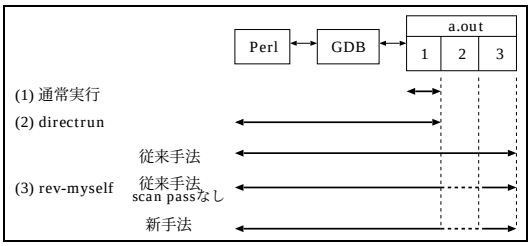

図 8 各テストケースの処理の内訳
Fig. 8 Details of test cases.

4.2 フック関数の実装

フック関数の実装を**図 6** に示す．3.1 節で述べたとおり，mcount1 ではスタックの深さとタイムスタンプをインクリメントして，シミュレート中のスタックに保存する．mcount2 ではスタックの深さをデクリメントする．

4.3 ユーザインタフェース

ユーザが利用する手段として，GDB のフロントエンドである mygdb[5] に機能付加する形で実装した．利用例を**図 7** に示す．図の状況は，関数 read_pattern_space 内をデバッガで実行しているところであり，下 2/3 程度に実行中のソースコードが表示されている．図中の下部で now というコメントで示してある網かけの部分が，デバッガがこれから実行しようとする行を示している．ここで関数単位疑似逆実行を指示すると，コメント target point で示した場所，すなわち現在実行中の関数である read_pattern_space の先頭に制御が移動する．

実際にデバッガの一機能として利用される場合，ユーザが設定したブレークポイントなどにより，単純に再実行を行ってもデバッギが期待どおりの場所で停止するとは限らない．また，頻繁にブレークポイントにヒットしてしまうと再実行の際の実行速度が極端に低下する．

そこで，再実行を行う前にユーザが設定したブレークポイントをいったんすべて無効（disable）にし，逆実行が終了した後に再び有効（enable）にする，という方法をとっている．

5. 経過時間計測

以下の各処理について，bash のビルトインである time コマンドで経過時間（elapsed time）を計測した（**図 8**）．

(1) プログラムを先頭から最後まで通常実行．
(2) gdb を起動してデバッギの実行を開始し，先頭から（次で行う）逆実行の目標点まで実行（directrun）．
(3) gdb を起動してデバッギの実行を開始し，プログラムの最後の関数呼び出しの入口で疑似逆実行を指示，目標点（現在実行中の関数呼び出しの入口）で制御を止める（rev-myself）．

第 1 の場合のみ，GCC の -O3 オプションで最適化した場合の時間も示す．第 2，第 3 の場合はデバッガとのインタラクションが必要となるが，Expect モジュールを用いた Perl スクリプトによってこれを自動化し

表 1 hanoi の処理時間
Table 1 Time spent on processing hanoi.

	従来手法	新手法	最適化
通常実行	20.920	19.236	8.13
directrun	30.166	28.688	—
rev-myself	80.968	50.488	—
scan pass なし	52.998	—	—

単位は秒

表 2 sed の処理時間
Table 2 Time spent on processing sed.

	従来手法	新手法	最適化
通常実行	4.816	4.854	3.088
directrun	14.126	14.114	—
rev-myself	28.574	21.136	—
scan pass なし	21.194	—	—

単位は秒

た．第 2 の場合は，その Perl スクリプトおよびデバッガの起動によるオーバヘッドを示すためのものである．第 3 の場合は，従来手法では通常実行，scan pass，re-execute pass と，デバッギが 3 回実行されるが，新手法では通常実行と再実行の 2 回しか実行されない点に注意されたい．なお，従来手法については後の結果解析のために，scan pass を行わない場合の時間も測定した☆．

計測の対象として，関数呼び出しが処理の本質である hanoi と，現実的な規模のプログラムとして sed-1.18 を用いた．

hanoi では 23 枚の円盤で処理を行い，8388607 回目の関数呼び出しで停止させ，その呼び出しの入口へ逆実行した．これは最後の関数呼び出しで，そのときのスタックの深さは 24 になっている．

sed では 10816 行（1379312 バイト）のテキストファイル☆☆に対して`s/make/MAKE/g` を行い，`/dev/null` に出力した場合について，126471 回目の関数呼び出しでの逆実行を計測した．これも最後の関数呼び出しで，そのときのスタックの深さは 2 になっている．

それぞれの計測を 5 回ずつ行い，平均した結果を**表 1** と**表 2** に示す．なお，表に掲載した時間は time コマンドの出力のうちの，elapsed time であり，計測に用いた計算機は IBM PC/AT 互換機，Linux-2.0.35，i486 DX2 50 MHz，メモリ 24 MB という構成である．

また，計測結果を差引きすることによって求めたテストケース各部の実行時間は**図 9**，**図 10** のようである．

☆ この実験では，開始時から逆実行目標点のタイムスタンプが分かっているので，scan pass を不要にできる．
☆☆ Linux カーネルを make した際のログファイルを連結したもの．

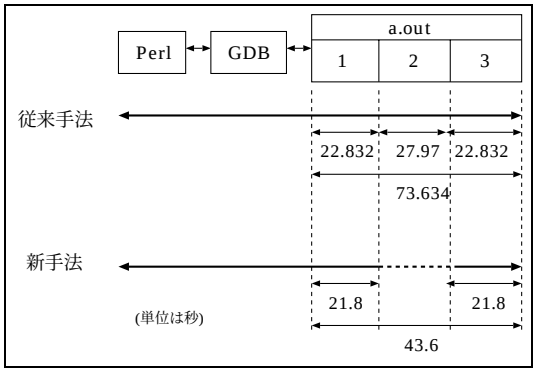

図 9 各手法の処理時間の内訳（hanoi）
Fig. 9 Result of each method in detail (hanoi).

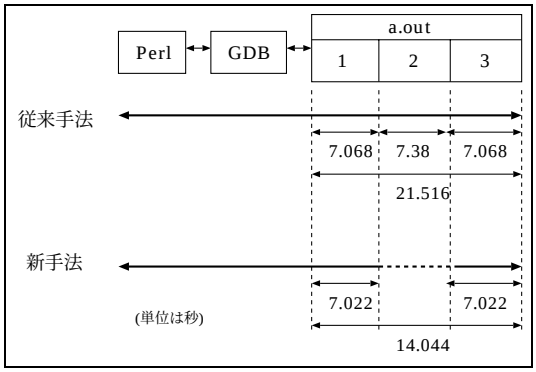

図 10 各手法の処理時間の内訳（sed）
Fig. 10 Result of each method in detail (sed).

5.1 考　　察

hanoi では表 1 に示したように，デバッギの実行回数は従来手法から新手法に変わって 2/3 になったのに対して，図 9 に示したデバッギの正味の処理時間はおよそ 0.59 倍になっている．一方 sed では，図 10 にあるようにおよそ 0.65 倍である．この性能向上率の違いは，hanoi の方がスタックの深さが深いため，従来手法の scan pass でデバッガが where を実行する回数が多いためで，それは図 9，図 10 で確かめられる．scan pass での where の実行は，OS，デバッガ，デバッギの間でタスク切替えが発生するため，速度低下の要因となる．実際，where の実行回数は hanoi が 23 回，sed が 6 回であった．

通常実行においては，関数呼び出しの入口と出口でフック関数を呼び出すために，新手法が従来手法に比べて 3/2 倍の関数呼び出しがある．しかし，フック関数の処理自身は従来手法の mcount に比べて単純化されている．これらの要因のトレードオフにより，従来

手法と新手法のオーバヘッドが hanoi と sed とで逆転していることの原因と考える．

6. ま と め

本論文では逆実行の実現方式として，これまで行われてきた履歴保存型とは異なる，先頭からの再実行を用いた疑似逆実行方式について，従来手法の特徴と問題点を述べた．従来手法では，ポータブルかつ簡易な実現を目指したが，一方では逆実行に 2 パスを要するという問題点をかかえていた．

そこで，その問題点を解決する新手法として，関数呼び出しの出口をも捕捉する手法を提案し，Intel Architecture，Linux，GCC という環境での実装について述べた．また，処理時間を計測した結果を示し，高速化の要因は再実行回数の減少だけでなく，デバッガ介入回数にも起因していることを述べた．

最後に，次章では本論文で提案した新手法の問題点と，今後の課題について述べる．

7. 今後の課題

新手法は従来手法に比して，以下のような欠点を持つ：

- アーキテクチャ，OS，コンパイラに依存した実装になっている．
- setjmp/longjmp に対応できない．
- mygdb という限られたインタフェースしか用意されていない．

第 1 点は，GCC の中間コードレベルで実装することでアーキテクチャと OS への依存度は解消できるが，コンパイラへの依存度は残ってしまう．第 2 点は，setjmp を利用するプログラマがソースコードを変更して，スタックの深さ（図 6 の depth）をも退避することで，mcount2 で捕捉できない関数からのリターンにも対応できる．第 3 点の改善としては，Emacs の gud-mode.el のような一般に広く使われているインタフェースに実装することで，解消したい．

参 考 文 献

1) Rosenberg, J.B.: *How Debuggers Work*, John Wiley & Sons (1997). 吉川邦夫（訳）：デバッガの理論と実装，アスキー出版局 (1998).
2) Lieberman, H. and Fry, C.: ZStep 95: A Reversible, Animated Source Code Stepper, *Software Visualization: Programming as a Multimedia Experience*, Stasko, J., Domingue, B.P. J. and Brown, M. (Eds.), MIT Press (1997).
3) 寺田 実：デバッガのためのプログラム疑似逆実行方式，情報処理学会論文誌，Vol.41, No.7, pp.1956–1963 (2000).
4) Biswas, B. and Mall, R.: Reverse Execution of Programs, *SIGPLAN Notices*, Vol.34, No.4, pp.61–69 (1999).
5) 寺田 実：気くばりデバッガ，秋のプログラミングシンポジウム「日本のプログラミング」報告集，中島秀之（編），pp.45–52, 情報処理学会 (1999).

(平成 11 年 12 月 29 日受付)
(平成 12 年 5 月 17 日採録)

丸山 一貴（学生会員）

1975 年生．1999 年東京大学工学部機械情報工学科卒業，同年同大学院工学系研究科情報工学専攻修士課程入学．現在，同修士課程在学中．プログラム言語処理系やユーザインタフェースに興味を持つ．ACM 学生会員．

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へて 1991 年東京大学工学部機械情報工学科講師，1992 年同大学助教授，現在に至る．工学博士．主な研究分野はプログラム言語処理系，プログラミング環境等．ソフトウェア科学会，ACM 会員．