情報工学専攻修士論文

プログラム実行点の特定と応用に関する研究

氏名：丸山 一貴

指導教官：寺田 実 助教授

2001 年 1 月 30 日提出

要旨

デバッグ作業の自動化を進めるためには、対象プログラムの実行を自動的に制御できなければならない。そのためには「ファイル名」と「行番号」と言ったような静的な情報では不十分なので、プログラムの実行履歴中の 1 点を特定できるような「動的な位置」の概念を提案する。これは静的な情報である (file:line) に、動的な情報である、ある種のカウンタの値を組み合わせて、(file:line, timestamp) と表されるものである。我々はこの動的な位置を position と名付け、これを用いてデバッグ作業がかなり自動化されるいくつかの応用を提案する。また、このカウンタ更新のためのコードを、対象プログラムのコンパイル時に自動的に組み込むために行った GCC への変更についても述べる。

目次

1 序論 **7**

1.1 背景 7

1.1.1 デバッガ 8

1.1.2 デバッガを用いたデバッグの実際 9

1.1.3 実行制御の困難 9

1.2 目的 10

1.3 本論文の構成 10

2 関連研究 **12**

2.1 逆実行に関する研究 12

2.1.1 状態保存による逆実行 12

2.1.2 再実行による逆実行 13

2.2 イベント駆動型実行制御 14

3 プログラム実行点の特定 **16**

3.1 静的な位置と動的な位置 16

3.2 動的ブレークポイント 18

3.2.1 動的ブレークポイントの実装 18

3.3 タイムスタンプの更新 19

4 タイムスタンプ機構の実装 **21**

4.1 実装の方法 21

4.2 タイムスタンプ更新コード 22

4.3 GCC の変更点 23

4.3.1 構文解析器への変更 24

4.3.2 アクションへの変更 24
4.3.3 コンパイラドライバへの変更 27
4.3.4 追加されるコード 28
4.4 デバッギの実行速度 28

5 実行点特定の応用 **30**
5.1 関数単位疑似逆実行 30
5.1.1 従来手法 31
5.1.2 従来手法の欠点 32
5.1.3 新手法 33
5.1.4 本論文での位置付け 35
5.2 実行点の印づけ 35
5.3 逆向きウォッチポイント 36
5.3.1 利用例 37
5.3.2 逆向きウォッチポイントの実装 40
5.3.3 逆向きウォッチポイントの処理時間 41
5.4 タイムスタンプを用いた実行履歴中の 2 分探索 42
5.4.1 2 分法のアルゴリズム 43
5.4.2 2 分法と `assert` との比較 44

6 各種の議論 **46**
6.1 履歴保存型逆実行 46
6.2 イベント駆動型実行制御 47
6.3 タイムスタンプの更新間隔 48

7 結論 **49**
7.1 結論 49
7.2 今後の展望 49

謝辞 **51**

参考文献 **52**

A GCC への修正 **54**

A.1 タイムスタンプ更新コード用初期化手続き 54

A.2 タイムスタンプ更新コード挿入手続き 54

図一覧

2.1 ZStep95[LF97] 13
2.2 疑似的な逆実行 14
2.3 Ducassé[Duc98] における実行制御コマンド 15

3.1 ループ構造を持つデバッギ 16
3.2 ループ構造を持つデバッギの実行履歴 17
3.3 location と position 18
3.4 動的ブレークポイントの実行手順 19

4.1 コンパイラの構成 22
4.2 タイムスタンプ更新コード 23
4.3 構文解析器の振舞 24
4.4 ループ構造の RTL への展開 25
4.5 追加されるアセンブリコード（i386-linux （左）と sparc-sunos4 （右）） 28

5.1 親関数の入口 31
5.2 シミュレートするスタック 32
5.3 従来手法 [Ter00] 32
5.4 関数呼び出しの出口のトラップ 33
5.5 各手法の処理時間の内訳 35
5.6 `simple_alloc` による領域管理 37
5.7 メモリ確保プログラム 38
5.8 本来のヒープの構造 39
5.9 破壊されたヒープ 39
5.10 逆向きウォッチポイントのアルゴリズム 41
5.11 2 分法の例 44

A.1 関数 `init_for_inc_ts` 55
A.2 関数 `emit_inc_ts` 56

表一覧

2.1 Inverse Statement[BM99] 13

4.1 タイムスタンプ機構のオーバヘッド 29

5.1 新手法と従来手法による `sed` の処理時間 [MT00] 34
5.2 逆向きウォッチポイントの処理時間 42

6.1 IGOR における保存量の低下 [FB88] 47
6.2 Boothe の方法での順方向コマンドのオーバヘッド [Boo00] 48

第 1 章

序論

1.1 背景

プログラムにバグはつきものであり、ソフトウェア開発において投入すべきリソース（マンパワー、時間、コストなど）のうちの 45% を、デバッグ・検査工程が占めている [Ish86]。さらに、デバッグ工程の結果は出荷品質を直接左右し得るため、効率的なソフトウェア開発のためには、デバッグ工程にかかる時間とプログラマの労力を削減することが不可欠である。

もちろん、バグのほとんどはプログラマが論理的に正しくないコードを書いてしまったことに起因しているので、自分の書いた全てのコードが論理的に正しいかどうかを検証すれば良い。しかしながら大規模なプログラムは、たとえそれが自分で書いたものであっても、全てを精読して検証することは不可能である。

このため、最初からバグを混入しないようにプログラミングすることに重きを置いた手法がある。仕様書を記述する際に数学記法を用いて形式的に記述することにより、自然言語による曖昧さを排除しようというものである [Inc92]。あるいは、意味論を用いて自動的に検証する手法もある [Mey91]。しかしながら、これらの手法はいずれも実用レベルでは未だ成功しているとは言い難い。

人手によるデバッグにおいてはプログラム全体を検証することは困難なので、プログラマは、バグの結果からその原因となる仮説を案出して対応するコードを検証する、というサイクルを繰り返して原因を特定することになる [Mye79]。検証の際にはプログラムの実行時の状態を正しく把握する必要があり、静的な情報であるコードだ

けでは不十分である。そこで、実行時のプログラムの状態を取得することが必要になる。

1.1.1 デバッガ

実行時のプログラムの状態を取得する最も単純な方法として、コード中に印字命令を挿入する方法（以下、**printf** デバッグと呼ぶ）がある。プログラム中の随所に C における **printf** のような、変数の値を表示する命令を追加して再コンパイルし、プログラムがどこから期待通りに振舞わなくなっているかを調べることができる。これはあまりにも安直な方法であり、適応できるのはかなり単純で小規模なプログラムの、ごく簡単なバグを取る時だけである。大規模なプログラムや複雑なバグに対してこの方法を行うと、大量の **printf** 文を挿入して様々な変数の値を表示する必要が起こる上に、膨大な量の表示を解析しなくてはならない。

現在は、デバッグ対象プログラム（以下、デバッギと呼ぶ）の状態をより効果的に取得するため、デバッグを補助する専用のツール、デバッガが用いられる。デバッガは非常に豊富な機能を有するが、主に利用されるのは以下のようなものである。括弧内に、代表的なデバッガである GDB[SPS+00] での対応するコマンド名を示す。

- デバッギの状態を検査するコマンド
  - 変数の値を表示（**print**）
  - 変数の値を変更（**set**）
- デバッギの実行を制御するコマンド
  - 実行をコード行単位で進める（**step**、**next**）
  - 特定のコード行に制御が到達したらデバッギの実行を停止する（**break**）
  - 特定の変数（メモリ領域）にアクセスがあったらデバッギの実行を停止する（**watch**）

これらのコマンドは主に 2 つに分けられる。すなわち、デバッギの実行時の状態を検査するコマンド（状態検査コマンド）と、デバッギの実行を制御するコマンド（実行制御コマンド）である。デバッガを利用するプログラマは、これら 2 種類のコマンドを相補的に利用してデバッグを行う。

1.1.2 デバッガを用いたデバッグの実際

デバッギの状態を詳細に制御、検査するツールはデバッガをおいて他になく、デバッグをするときには簡単なものを除いてはデバッガを用いて行うのが一般的である。従って、現在のデバッグ作業はデバッガの能力に大きく依存しており、主に 3 つの作業を行うことになる。

1. エラーの原因となりそうなソースコード中の場所を探し、エラー原因の仮説を設定する
2. 設定した仮説に基づいて、原因と思われる場所まで実行制御コマンドでデバッギの制御を移す
3. 状態検査コマンドでデバッギの状態を調べ、仮説を検証する

第一の作業は最も重要な作業であり、デバッグの本質である。第三の作業は第一の作業における仮説を確かめるために必要な作業である。しかし、第二の作業はデバッグにとって本質的な作業ではなく、既存のデバッガの能力のためについてまわってしまう。また、この作業は特に苦痛を伴う作業であり、目標とする実行状態をうっかり通り過ぎてしまったらデバッギを先頭から再実行して、同じ作業を繰り返さなくてはならない。このように、実行制御はデバッグに要する時間の大半を占める作業である。しかも、第一の作業で設定した仮説が誤っていた場合、新たな仮説の検証のために、制御を再び別の点へ移動させることになる。

1.1.3 実行制御の困難

実行制御が困難な理由は 2 つある。第一に、特定の行で制御を停止する仕組み（以下、ブレークポイントと呼ぶ）が静的な情報に基づいているということである。デバッガの実行制御コマンドには、GDB における `break` や `watch` のようなイベント駆動型の機構がある。この仕組みを用いると、デバッギの制御が目標の行に到達したり、監視対象の変数の値が変更されるなどのイベントが発生しない限り、デバッギは通常速度で実行される。一方、ステップ実行を繰り返し行うのは非常に低速なので、制御を目標の場所に素早く移動させるにはブレークポイントをうまく利用することが重要になる。しかし、ブレークポイントは静的な情報であるファイル名と行番号を対

象として設定されるので、デバッギは本当の目標点に到達する前に何度も停止してしまう。プログラマはその都度、変数の値を詳細に調べ、本当の目標点で停止したのかどうかを判断しなければならない。この問題の解決のためには、動的な情報を含む新しいブレークポイント機構が必要である。

第二に、「動的な実行点の位置」を指定する方法がない、ということである。仮に、デバッギの実行開始と同時に目標の行にブレークポイントを設定できていれば、そのヒット回数を動的な位置として用いることができる。しかしデバッグ中のプログラマがブレークポイントにヒットした回数を、しかも各対象行ごとに管理するのは非常に難しい。この解決のために、ブレークポイントのヒット回数といった生々しい情報を覆い隠し、デバッギの実行中に一貫性を保つような「動的な位置」と、その位置に印づけをする仕組みが必要である。

また、これらの仕組みが出来上がれば、近年の計算機の強力な処理能力を利用して、現在は完全に手動で行っているデバッギの実行制御をかなり自動化することができるようになる。バグの原因の場所を突き止めるといったような知的な処理を多少は自動化する有用なアプリケーションを作ることもできる。これらの方法はひどく乱暴なもののようではあっても、プログラマはデバッガを制御する苦痛から解放されてデバッグそのものに集中できるようになる。

1.2 目的

本研究では再現性のある C プログラムを対象とし、先頭から再実行することによって目標となる動的な位置へ自動的に制御を移動することを目的とする。さらに、そのいくつかの応用を示すことで新しいデバッグ手法を提案する。また、これらの仕組みを既存のコンパイラとデバッガを用いて実装する方法についても述べる。

1.3 本論文の構成

本論文は「プログラム実行点の特定と応用に関する研究」と題し、以下の各章からなる。

第1章は「序論」である。

第2章は「関連研究」と題し、プログラムの実行制御に関する先行研究について述べる。

第3章は「プログラム実行点の特定」と題し、ソースコードにおける位置である location と、プログラム実行履歴における位置である position を提案する。

第4章は「タイムスタンプ機構の実装」と題し、position を実現するために必要となるタイムスタンプ機構を対象プログラムに組み込むための、GCC の変更点について述べる。また、position を対象として設定する動的ブレークポイントを提案し、その実装についてもあわせて述べる。

第5章は「実行点特定の応用」と題し、position を用いることによって自動化されるデバッグ手法を提案する。

第6章は「各種の議論」と題し、主に先行研究と本研究との比較を行う。

第7章は「結論」である。

付録Aは「GCC への修正」と題し、実際に GCC に追加した主なコードを示す。

第 2 章

関連研究

本章では、実行制御に関する先行研究として、逆実行に関する研究とイベント駆動型実行制御について紹介する。本研究と各研究の比較は第 6 章で行う。

2.1 逆実行に関する研究

ここでは主要な 2 種類の逆実行方式について述べる。

2.1.1 状態保存による逆実行

逆実行を完全に実現するには、代入によって失われる情報を全て保存していけば良い。このような考えに立って、プログラムの実行状態の履歴を保存していくという方法で逆実行を実現した研究は多い [LF97, Moh88, FB88, ADS91, BM99]。これらは皆、専用の処理系ないしデバッガを実装し、そこに状態を保存するための機構を組み込むことで逆実行を実装している。

Lieberman ら [LF97] （図 2.1）と Moher[Moh88] は、かなり単純にプログラムの実行状態全ての保存を行っているが、これらの方法では実行履歴が余りにも大きくなってしまうので、他の研究ではそれぞれに履歴を小さくするための工夫を行っている。

Feldman ら [FB88] は、仮想記憶のページの中で前回の保存以降に書き込みのあったページだけを保存するという方法を用いている。 Agrawal ら [ADS91] は、動的プログラムスライシングを用いてあるステートメントが変更し得る変数群だけを保存

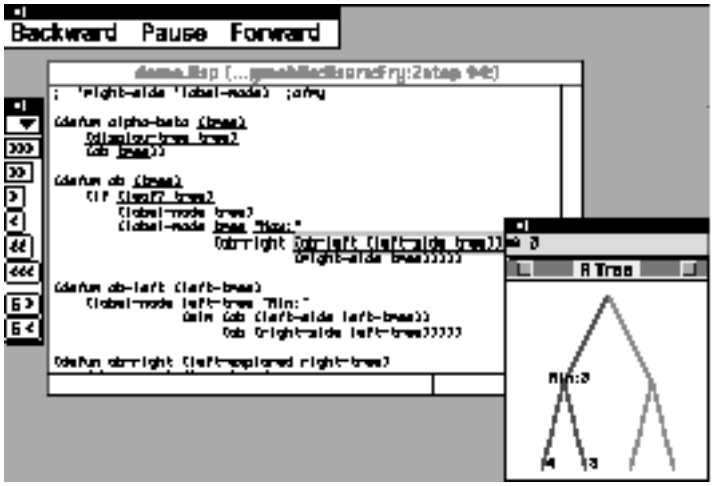

図 2.1: ZStep95[LF97]

表 2.1: Inverse Statement[BM99]

Original Statement	Inverse Statement
`a += b;`	`a -= b;`
`i++;`	`--i;`
`b += c++ + 1;`	`b -= --c - 1;`

することで、履歴の量を減らそうとしている。 Biswas ら [BM99] は、逆演算を行う “inverse statement” を用いることで静的に推測できる情報は保存を行わないようにしている。 inverse statement の例を表 2.1に示す。

2.1.2 再実行による逆実行

一方で、近年の計算機の高速化を利用して、先頭から現在点の直前まで再実行することによって疑似的に逆実行をする手法が提案されている。これまでの逆実行方式では現在点から一歩手前まで直接戻ることを目指していたが、疑似的な逆実行では先頭から順方向に再実行してその場所を目指すので、履歴を保持する必要がなくなる。その代わり、再実行パスで目標点となる「一歩手前」の場所を識別できなければならない（図 2.2中の target）。この識別のために専用の特別なカウンタを用意することになる。

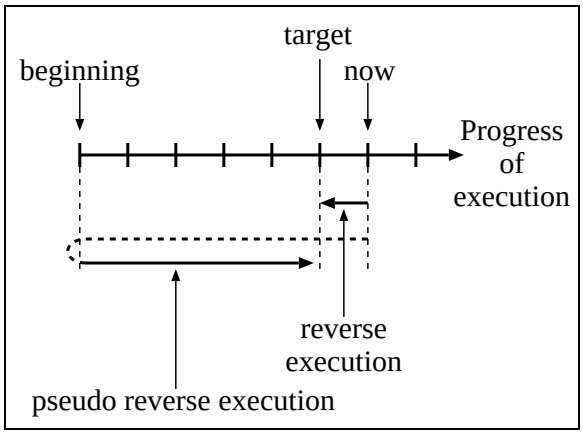

図 2.2: 疑似的な逆実行

寺田 [Ter00] は、プロファイル用関数[1]を再定義して、現在実行中の関数の入口を目標点とする、関数単位の逆実行を容易かつポータブルに実装することに成功している。 Boothe[Boo00] は、デバッガの持つ全ての実行制御コマンドの逆実行版を実装することで、逆実行の機能を持ったデバッガを構築することを目指した。ステートメント単位の逆実行が必要になるため、カウンタとしてデバッガで行うステップ実行の回数を用いている。本研究では、こうしたカウンタを用いた目標点特定の新たな手法を提案する。

2.2 イベント駆動型実行制御

Ducassé[Duc98] は、デバッギの実行を従来のステートメント指向で制御するのではなく、代入や関数呼び出し、ループといった、イベント指向で制御するためのデバッガを実装している。プログラマは Prolog に似た書式で複雑な条件を持つブレークポイントを設定できる。例えば、図 2.3のようなコマンドをデバッガに与えると、「関数 `legal` の中で `for` 文に入るか `if` 文から出るようなときにだけ実行を停止せよ。`for` のときであれば盤面を表示し、変数 `lineNb` と `colNb` も表示せよ。それ以外のときは変数 `i` と `board(I)` と `ok` を表示せよ。」という指示をしたことになる。

組み込みのコマンド（`fget` や `if` など）を組み合わせてユーザが新しいコマンドを定義できるようになっており、図 2.3で登場する `display_board` はこれにあたる。ま

[1]プログラムの性能測定のために、関数呼び出しの入口で呼び出される関数。この呼び出しはコンパイラによって自動的に挿入される。

```
fget(func=legal and
            ((type=for and port=enter) or
             (type=if  and port=exit))),
  ( if   current(type=for)
    then display_board,
         current_var(lineNb, val=Lin),
         current_var(colNb, val=Col)
    else current_var(i, val=I),
         current_var(board(I), val=BdI),
         current_var(ok, val=Ok)).
```

図 2.3: Ducassé[Duc98] における実行制御コマンド

た、与えたコマンドによって制御が一旦停止しても、そこが目的の場所でなければ、続行を指示して条件が満たされる次の場所を探すことができる（バックトラック）。

第 3 章

プログラム実行点の特定

前述のように、再実行による逆実行を実現するためには目標点を特定するための専用のカウンタが必要になる。本章ではこれを提案する。

3.1 静的な位置と動的な位置

目標点を特定するために、デバッギが特定の実行状態に到達した時に自動的に通知されるような仕組みが望ましい。通常、こうした目的のためにはデバッガが持つブレークポイント機構が利用されるが、これは今回の目的にはそぐわない。

図 3.1のようなループ構造を持つデバッギがあるとする。ループの 3 回目のときの行 (2) が目標点であるときにブレークポイントを行 (2) に設定しても、先頭から再実行するとそのブレークポイントは目標点に到達する前に、目標点とは異なる場所で 2

```
:
(1) while(i < a){
(2)   i += b;
(3) }
:
```

図 3.1: ループ構造を持つデバッギ

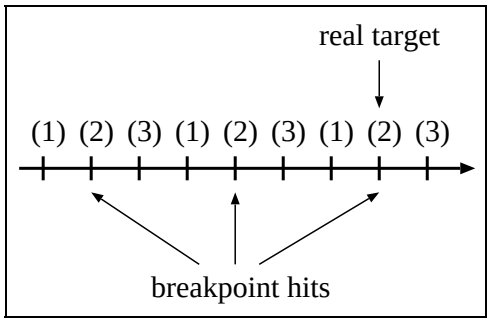

図 3.2: ループ構造を持つデバッギの実行履歴

回ヒットすることになり、何回目のヒットが真の目標点であるかを知ることができない（図 3.2）。

ブレークポイントが不十分な理由は、それがソースコード中の位置、すなわち「ファイル名」と「行番号」という静的な情報に基づいて設定されるからである。そこで、これを静的な位置と考え、location と呼ぶことにする。これは (file:line) で表される静的に unique な情報で、ソースコード中の一点を表すことができる。図 3.1の行 (2) の location は、ファイル名を test.c とすると (test.c:2) として表される。

これに対して図 3.2での真の目標点は、3 回目に登場する行 (2) である。何回目に登場する行 (2) が真の目標点であるかは実行中のプログラムの持つ動的な情報に基づいて決定されるので、これを動的な位置と考え、position と呼ぶことにする。これは動的に unique な情報で、プログラム実行履歴中の一点を表すことができる。

一般に、1 つのステートメントは 1 つの location に対応し、1 つ以上の position に対応する[1]。そこで、実行履歴中に同じ location が登場する際に必ず更新されるようなカウンタを用いて目標点を識別する。この適切なタイミングで更新されるカウンタをタイムスタンプと呼び、デバッギ中に大域変数 `timestamp` として導入すると、position は (file:line, timestamp) のように表せる。今の例で、目標の position での `timestamp` が 8 であったとすると、目標点は (test.c:2, 8) と表せる。一方、それ以前に登場する 2 回の (test.c:2) はそれぞれ (test.c:2, 6) と (test.c:2, 7) ということになる（図 3.3）。position は location に動的な情報であるタイムスタンプを付加したものと考えられる。

[1]ただし、そのステートメントが条件分岐で選択されない部分にあるなどして実行されない場合には、対応する position は存在しないので 0 個となる。

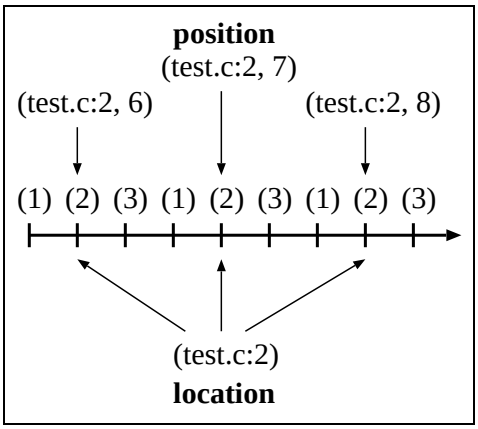

図 3.3: location と position

3.2 動的ブレークポイント

通常のブレークポイントは location、すなわち (file:line) に基づいて設定するので、これを静的ブレークポイントと呼ぶことにする。前述の通り、プログラム実行履歴では location は複数回登場し得るので、静的ブレークポイントは複数回ヒットしてしまう（先ほどの例では 3 回）。

そこで、 position に対して設定できるブレークポイントを考え、これを動的ブレークポイントと呼ぶ。動的ブレークポイントは (file:line, timestamp) を指定して設定することになる。

3.2.1 動的ブレークポイントの実装

動的ブレークポイントを実現するには、既存のデバッガと同様の仕組みが必要になる。デバッガが動作するにはハードウェアや OS による支援が不可欠なので [Ros97]、既に多くのプラットフォームで動作実績のあるデバッガを利用するのが便利である。そこで、以下では GDB が持つ機能が利用できるものとして説明を行う。

動的ブレークポイントの実装は次のようになる。まず、タイムスタンプ値が目標点のそれと等しくなる更新位置まで実行する（step 1）。すると、そこから目標の position までの間に、目標点と同じ location が別の position として存在し得ないことが保証されているので、通常のブレークポイントを用いて、さらに真の目標点まで実行すれば良い（step 2）。

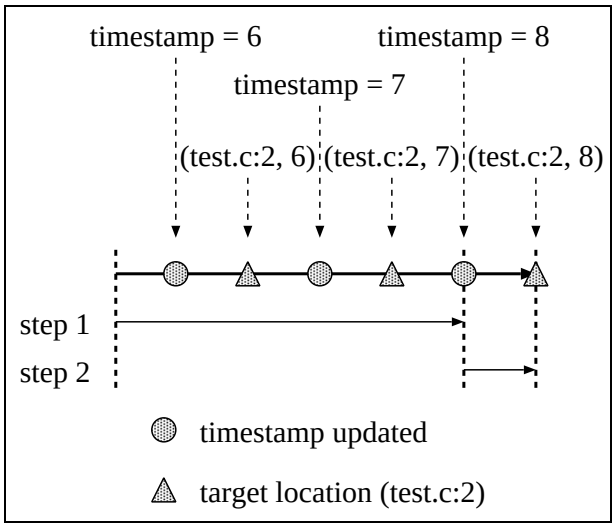

図 3.4: 動的ブレークポイントの実行手順

図 3.1におけるタイムスタンプ更新のタイミングを図 3.4に示す。この例では目標点と同一の location は実行履歴中に 3 回登場する（図中の三角印）。しかしタイムスタンプが更新されるタイミングは、隣り合う location の間に必ず 1 つ存在する。すなわち、目標 position のタイムスタンプ値は目標 position の直前に存在し、その点から目標 position の間には同一の location は存在しない[2]。従って、前述の step 1 と step 2 は図のようになる[3]。

3.3 タイムスタンプの更新

実行履歴中に location が繰り返し現れ得る場合とは、関数呼び出しが起こる場合と、同一関数内でプログラムの制御が逆方向へ不連続にジャンプする場合である。従って、タイムスタンプは関数呼び出しと上向きのジャンプが起こり得るときに更新すれば良いので、以下のような場所に制御が到達したときにのみ行う。

[2] この場合の例では、タイムスタンプ更新点の間に必ず目標点と同一の location が出現しているが、出現しない場合ももちろん存在する。タイムスタンプは、その値が次に更新されるまでの間に、目標点と同一の location が 0 回または 1 回だけ出現することを保証する。

[3] 動的ブレークポイントはデバッガの持つ条件付ブレークポイントを用いても実装できる。対象の (file:line) に対して、タイムスタンプ値が timestamp であるという条件を付けて指定すれば良い。しかしながら、これは静的ブレークポイントに到達する度にデバッギからデバッガへのトラップが発生し、OS でのタスク切替が起こるので実行速度が非常に遅くなる。デバッギからのトラップを最小限にするためには目標のタイムスタンプ値になったときにだけトラップが発生することが望ましい。

- 関数呼び出しの入口・出口
- `return` 文
- ループの繰り返し
- 上にあるラベルへの `goto` 文によるジャンプ

関数呼び出しの入口での更新は、繰り返し呼び出される関数のそれぞれの呼び出しを区別するために必要となる。出口での更新が必要な理由は、再帰的に定義された関数では再帰呼び出しからリターンした後で同一の location を通過する可能性があるからである。`return` 文は関数呼び出しの出口と同様であるが、返り値を持つ場合があるので実装上は別のものとして扱わなければならない。

タイムスタンプの値は、プログラマがその値を推定して直接指定できるような性質のものではない。例えば「あのループの 325 回目に興味がある」と思っても、そのときのタイムスタンプ値を推定することは事実上不可能であり、デバッガを用いてその position まで到達して変数 `timestamp` の値を取得するしか方法はない。タイムスタンプ値は、第5章で述べるような応用を実現するプログラムから利用することを想定しており、プログラマが直接利用することは想定されていない。

第 4 章

タイムスタンプ機構の実装

本章では、前章で述べたタイムスタンプ更新機構をデバッギに自動挿入するための方法について述べる。

4.1 実装の方法

デバッギにタイムスタンプ機構を挿入するには、以下の 3 つのレベルが有り得る。

1. ソースコードレベル
2. 中間コードレベル
3. アセンブリコードレベル

第 1 のソースコードレベルとは、ソースコード変換を用いて行う方法である。この方法はソースコードそのものを対象とするので、処理系や OS、プラットフォームを選ばないという利点があるが、デバッグ中にプログラマから見えるコードは変換後のコードになってしまうという欠点がある。

第 2 の中間コードレベルでは、コンパイラが内部的に持っている中間コードを変換してタイムスタンプ機構を挿入する。コンパイラはソースコードを入力として受け取ると一旦この中間コードに変換し、最適化などを施した後で対象プラットフォームのアセンブリコードとして出力する（図 4.1）。例えば、GNU C Compiler[Sta99] では Register Transfer Language （RTL）と呼ばれる中間コードを利用している。中間

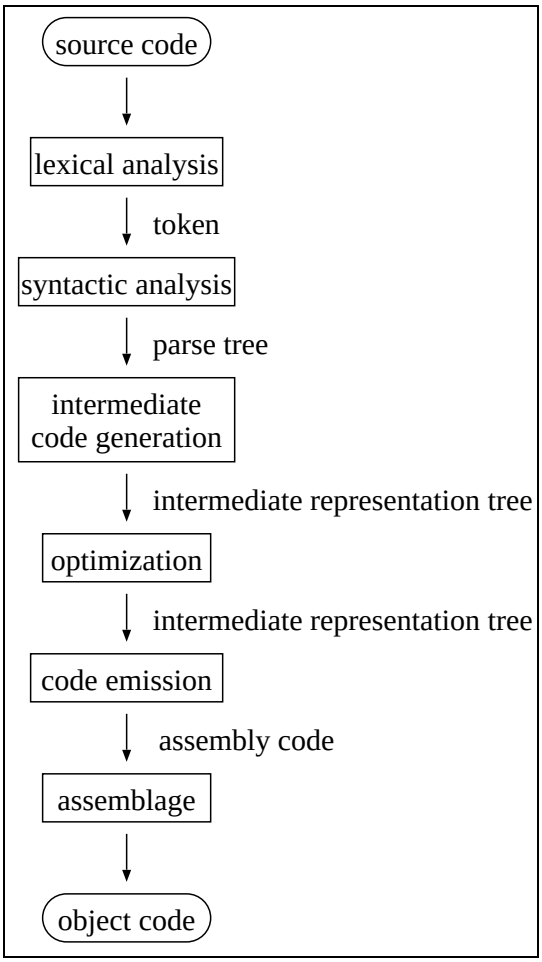

図 4.1: コンパイラの構成

コードレベルで挿入すればポータビリティが失われることはないが、特定の処理系に依存することになる。

第 3 のアセンブリコードレベルでは、図 4.1の assembly code を変換する。これは実装が最も容易な方法だが、各環境毎に実装する必要があるためポータビリティに欠ける。

以上より、第 2 の中間コードレベルで実装することとし、対象とする処理系は様々なプラットフォームで広く利用されていることと GDB との連係を考慮して、GCC を選んだ。今回は現時点での最新版である GCC-2.95.2 を実装の対象とした。

4.2　タイムスタンプ更新コード

```
int timestamp = -1;
int ref = -1;

void inc_ts_stop(void){}

#define INC_TS if(++timestamp == ref) inc_ts_stop();
```

図 4.2: タイムスタンプ更新コード

タイムスタンプ更新場所に到達したときに図4.2のタイムスタンプ更新マクロ **INC_TS** を実行するように、デバッギが変換される。デバッギからデバッガへのトラップが、特定のタイムスタンプ値に到達したときにだけ起こるように工夫してある。

動的ブレークポイントの step 1 では、タイムスタンプが目標の値になるまでデバッギの実行を進める。このとき動的ブレークポイントを実行する手順は以下のようになる。最初に関数 **inc_ts_stop** に静的ブレークポイントを設定する。次に変数 **ref** に目標のタイムスタンプ値を設定してデバッギの実行を開始する。 **INC_TS** が実行されるたびに変数 **timestamp** のインクリメントと **ref** との比較が行われ、タイムスタンプが目標値に達したとき（**timestamp** と **ref** の値が等しくなったとき）、関数 **inc_ts_stop** に設定した静的ブレークポイントでデバッギが停止する。デバッギはここまで通常速度で実行される。

デバッギはコンパイラによって、全てのタイムスタンプ更新場所にマクロ **INC_TS** を展開したコードの RTL を自動的に挿入される。さらに、このマクロを正常に機能させるために、デバッギの各ファイルの先頭部分には変数 **timestamp** と **ref** の **extern** 宣言、関数 **inc_ts_stop** のプロトタイプ宣言が RTL として挿入される。これらの実体は別途ライブラリとして用意し、コンパイラが自動的にリンクする。

4.3 GCC の変更点

図 4.1ではコンパイラの処理のステージを列挙したが、実際のコンパイラの処理は

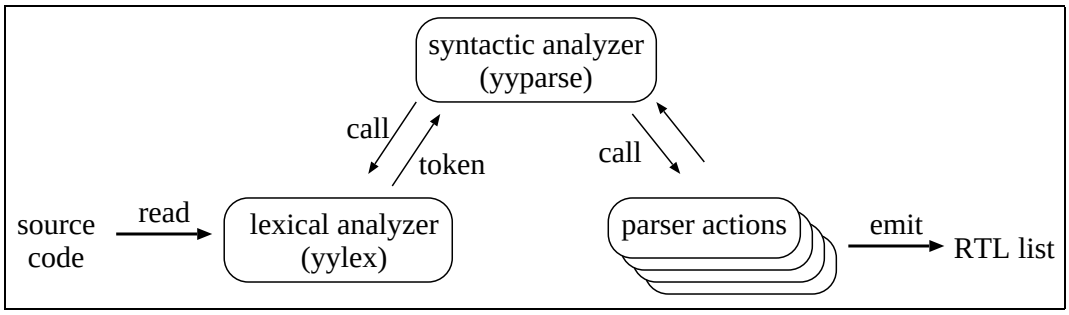

図 4.3: 構文解析器の振舞

明確には分かれていない。GCC は構文解析器である関数 **yyparse** が字句解析器である関数 **yylex** を呼び出し、読み込んだトークンを構文解析して対応するアクションを呼び出し、そのアクションが随時 RTL を出力するようになっている（図 4.3）。RTL は双方向リスト構造をしているので、ここでいう出力とはそのリスト構造へ新しい RTL 式を追加することを意味する。

我々が変更しなければならない部分は以下の 3 つに大別できる。以下の節でこれらを順に説明する。

- 構文解析器（**yyparse**）
- タイムスタンプ機構に関わるいくつかのアクション（ループのためのアクション、関数定義のためのアクション、など）
- コンパイラドライバに関わるいくつかの関数とマクロ

4.3.1　構文解析器への変更

構文解析器である関数 **yyparse** はコンパイラが新しいソースファイルを読み込むたびに呼び出されるので、その処理の最初に変数 **timestamp** と **ref** の **extern** 宣言、関数 **inc_ts_stop** のプロトタイプ宣言を RTL リストへ追加する。

4.3.2　アクションへの変更

前章で挙げた、タイムスタンプを更新すべき 4 つの場合のそれぞれについて、どのアクションのどの部分で **INC_TS** の RTL を挿入すれば良いかを述べる。

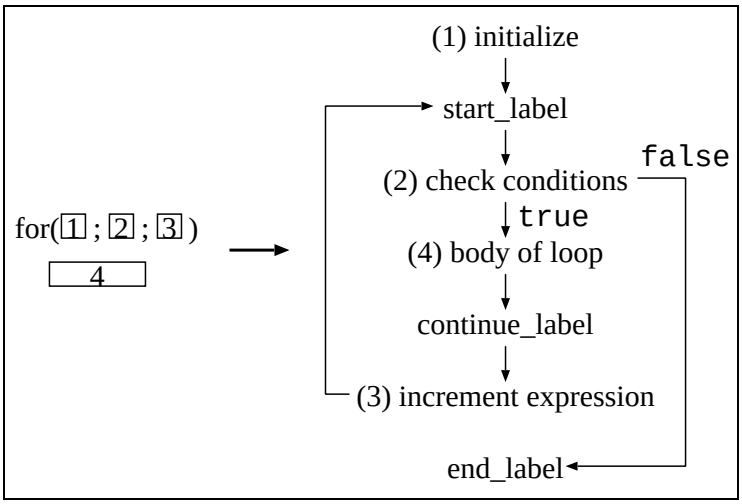

図 4.4: ループ構造の RTL への展開

ループの繰り返し

ループが繰り返される場合は2 通りある。 1 つはループ本体の末尾に到達した場合で、もう 1 つは **continue** 文が実行された場合である。これらを考慮して、 GCC はループを図 4.4のような RTL リストに展開する。 **start_label** はループ本体の末尾に到達したときのジャンプ先ラベルであり、 **continue_label** は **continue** 文が実行されたときのジャンプ先ラベルである。ここでは **for** 文を取り上げたが、 **do-while** 文でもほぼ同様になる。 **while** 文では特に、 **start_label** と **continue_label** が一致する。

タイムスタンプ更新場所であるループの繰り返しは本来ループ本体（図中の 4）の末尾に相当するが、 **continue** 文による繰り返しも一緒に扱うようにした方が実装が容易になるので、 **start_label** の直後で更新することにする。ループに入る最初の 1 回だけ余分に更新されることになるが、全体の計算量に比べれば無視できる量である。

while 文のときは、構文解析器はアクションである関数 **expand_start_loop** を呼び出す。 **for** と **do-while** のときは、 **expand_start_loop_continue_elsewhere** が呼び出され、これが **expand_start_loop** を呼び出す。 **start_label** の出力は **expand_start_loop** 内の関数 **emit_label** の呼び出しで行われるので、その直後で **INC_TS** を展開した RTL を RTL リストの末尾に追加した。

goto 文

ANSI C に基づく goto 以外にも、GCC 独自の拡張として nonlocal goto と computed goto がある。nonlocal goto は nested function において外側の関数内部のラベルへジャンプするものであり、computed goto はジャンプ先のラベルを変数の値を用いて決定する。いずれも使用頻度は低いと考えられるので、今回の実装では考慮しなかった。また、本来はジャンプ先のラベルが下方にある場合はタイムスタンプ更新は不要であるが、実装を容易にするためジャンプ先ラベルの位置の判定を行わないことにした[1]。

通常の goto 文と nonlocal goto の場合、構文解析器はアクションである関数 expand_goto を呼び出す。expand_goto は nonlocal goto かどうかを判定して、通常の goto の場合は直ちに expand_goto_internal を呼び出すので、この呼び出し直前で INC_TS を追加するように変更した。

return 文

return 文のときは構文解析器がアクション c_expand_return を呼び出すが、返り値がある場合とない場合とでその後の動作が異なる。返り値がない場合は expand_null_return が呼ばれてそこから expand_null_return_1 が呼ばれるが、返り値がある場合は expand_return 経由で expand_null_return_1 が呼ばれ、いずれの場合も最終的にはそこで関数定義のエピローグへ向かって expand_goto_internal によってジャンプすることになる。

INC_TS の追加は 2 つの場合が合流する関数 expand_null_return_1 で行うこともできるが、返り値がある場合、その段階では既にレジスタ内に計算済みの返り値が設定されており、そこでタイムスタンプ更新マクロを実行すると返り値を壊してしまうことになる。返り値が保存されている可能性のあるレジスタをスタックに退避してからタイムスタンプを更新するという方法もあるが、ここでは return 文の処理を開始する前である、c_expand_return の最初で追加することにする。

[1] タイムスタンプの更新は、実行履歴中に同一の location が再び登場するまでの間に少なくとも 1 回登場すれば良いので、更新し過ぎてしまっても position の妥当性は失われない。ただし、余分な更新を行うことで処理速度が多少犠牲になる。

関数呼び出しの入口

関数定義の先頭ではまず、関数名と型の情報などを登録する `start_function` が呼ばれ、次に引数の名前と型を登録する `store_parm_decls` が呼ばれ、定義本体である compound statement の処理に移る。

関数呼び出しの入口で、かつ、関数定義のときにのみ `INC_TS` が実行されるためには、compound statement の処理に入る前に追加されることが望ましい[2]。そこで、`store_parm_decls` の末尾で追加する。

関数呼び出しの出口

関数のエピローグ部分を処理するアクションは `finish_function` である。このアクションは、`return` 文が実行されたときのジャンプの目標となる `return_label` をエピローグ部分に設定する。`return_label` よりも下に `INC_TS` を追加してしまうと、`return` 文のときに余分なタイムスタンプ更新が発生するだけでなく、レジスタに保存された返り値を破壊する可能性があるので、`return_label` より前に追加されなくてはならない。従って、関数 `finish_function` の最初で追加する。

4.3.3 コンパイラドライバへの変更

必要なときだけタイムスタンプ機構が組み込まれるように、GCC に新しいオプション `-pg2` を追加する。新しいオプションを設定するには、構造体 `default_compilers` を修正して、コンパイラ本体である `cc1` のオプション設定に追加する。次に `cc1` の `main` 関数のオプション解析ループの中に追加して、`-pg2` が指定された時だけ特別なフラグを立てるようにする。前述の `INC_TS` 追加用のコードは、全てこのフラグが立っている時だけ動作するようにしておく。

次に、`inc_ts_stop` などの定義が含まれるライブラリを `-pg2` 指定時に自動的にコンパイラがリンクするよう、各プラットフォーム用のヘッダファイルに含まれる `LIB_SPEC` マクロを修正する。例えば i386-linux であれば `gcc/config/linux.h` に含まれている。

[2] 構文解析器による compound statement の処理は、関数定義に限らず `if` 文などで登場する場合も同様に行われるので、compound statement のアクションで追加を行うと不要なタイムスタンプ更新が増え、実行性能に悪影響を与えてしまう。

```
incl timestamp
movl timestamp,%eax
cmpl ref,%eax
jne .L6
call inc_ts_stop
```

```
sethi   %hi(_timestamp), %o1
ld      [%o1+%lo(_timestamp)], %o0
add     %o0, 1, %o0
st      %o0, [%o1+%lo(_timestamp)]
sethi   %hi(_ref), %o1
ld      [%o1+%lo(_ref)], %o1
cmp     %o0, %o1
bne     L21
mov     0, %l2
call    _inc_ts_stop, 0
```

図 4.5: 追加されるアセンブリコード（i386-linux （左）と sparc-sunos4 （右））

4.3.4 追加されるコード

GCC に対する変更は全部で 70 行程度であった。このような変更を施した GCC の動作を以下のプラットフォームで確認した。別のプラットフォームへ移植するために必要なことは、プラットフォームごとのヘッダファイルに含まれるマクロを新たに 1 行書き直すだけである。

- i386-linux
- sparc-sunos4
- alpha-linux

実際に追加された RTL によるアセンブリコードを、i386-linux と sparc-sunos4 のそれぞれについて図 4.5に示す。これらは、オプションとしてデバッグ用とタイムスタンプ機構導入用、最適化用である、`-g -O -pg2` をつけて生成したコードである。

4.4 デバッギの実行速度

表 4.1: タイムスタンプ機構のオーバヘッド

GCC options	Celeron 400MHz (linux)	Alpha 500MHz (linux)	SPARC station20 (SunOS)
-g -O (sec.)	2.57	4.64	17.44
-g -O -pg2 (sec.)	3.78	6.34	24.40
factor	1.47	1.37	1.40

タイムスタンプ機構が追加されると、ない場合に比べてデバッギの実行速度が低下する。これを動作を確認した 3 つの環境で計測した。計測の対象としたデバッギは gawk-2.15.6 で、 GNU time-1.7 で計測を行った。この結果を表 4.1に示す。速度低下の度合は、いずれも実用上問題ないと考える。

第 5 章

実行点特定の応用

position を用いて、実行履歴中の任意の 1 点を特定できるようになった。これを用いると 2 種類の応用が考えられる。 1 つは人間が手動で行っていた手続きを自動化するような応用であり、もう 1 つは、それをさらに進めて、手動では到底できないような手続きを行うものである。これらはひどく乱暴な手法に見えるかも知れないが、近年の高速な計算機を利用することで、人間が手動で行うよりもずっと楽にデバッグを行うための有用な手法である。

5.1 関数単位疑似逆実行

最初の応用として逆実行について説明する。

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。第 2.1.2節で述べた再実行による疑似逆実行は、プログラマが手動で行うこのプロセスを自動化していると考えられる。このような実行制御プロセスはデバッグとは本質的に関係のないものであり、これを自動化することはデバッグ時のプログラマの負担を軽減する。

そこで、デバッグ時に有用な実行制御として実装されたのが、寺田 [Ter00] による関数単位疑似逆実行である。これは逆実行の単位を関数呼び出しに定め、「親関数の入口」を逆実行の目標点とするものである。以下では、寺田の手法（以下、従来手法と呼ぶ）と、これを高速化するために関数呼び出しの出口をトラップした新手法 [MT00] について説明する。

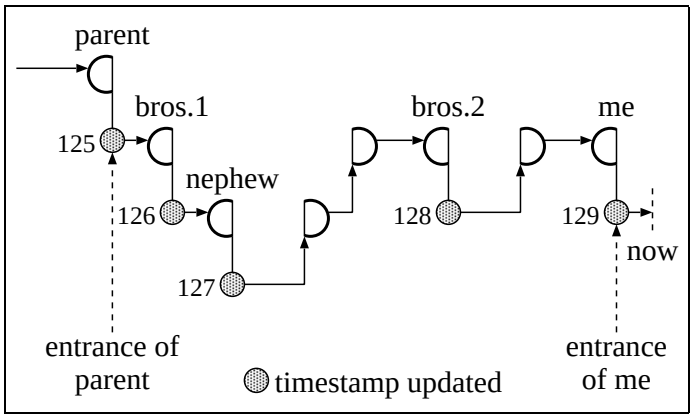

図 5.1: 親関数の入口

5.1.1　従来手法

第2.1.2節で述べたように、寺田はプロファイル用関数を再定義することで関数呼び出しが起こるたびに更新されるカウンタを用意し、これを用いて現在実行中の関数を呼び出した親関数の入口へと移動するという機能を実装した。これは人間が行うデバッグ作業において自然な場所である。ここでは便宜上、このカウンタもタイムスタンプと呼称して説明を行う。この場合のタイムスタンプは、関数呼び出しの入口でしか更新されない。

図5.1に例を示す。現在、関数 `me` を実行中とすると、自分（`me`）を呼び出した親関数は `parent` ということになり、その入口は図の通りである。図中の網かけの丸印はタイムスタンプ更新場所であり、左の数字がそのときのタイムスタンプ値を示す。自分の呼び出しは 129 であり、親の呼び出しは 125 である。

ここで問題となるのは、目標点である親関数の入口のタイムスタンプ値（ここでは 125）をどのようにして得るか、ということである。現在点でのタイムスタンプ値は `me` の入口で更新した値であり、目標点から現在点までの間に何回更新されたか、すなわち何回の関数呼び出しがあったかは一般には知ることができない。

関数呼び出しには必ず 1 つのタイムスタンプ値が対応するので、関数呼び出しのスタックをシミュレートして呼び出し深さごとの最新タイムスタンプを保持すれば良い。関数 `me` の呼び出しが深さ n とすると、シミュレートするスタックの深さ n に相当する場所にはタイムスタンプ値として 129 が保持される（図 5.2）。深さ $n-1$ には `parent` のタイムスタンプ値である 125 が、深さ $n+1$ には `nephew` が呼び出されたときの 127 が残っている。 `me` の親関数のタイムスタンプ値を知りたいときには、

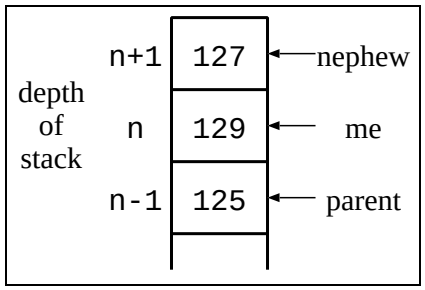

図 5.2: シミュレートするスタック

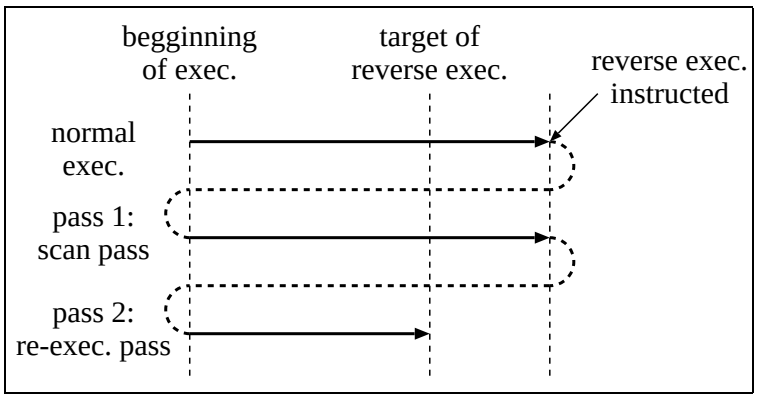

図 5.3: 従来手法 [Ter00]

自分の呼び出し深さよりも 1 つ浅い場所である $n-1$ を見ると、目標とする関数呼び出しのタイムスタンプ値が手に入ることになる。

5.1.2　従来手法の欠点

しかし、関数呼び出しの出口がトラップできないことから現在点の呼び出し深さが正確に分からないため、これを正確に知るための scan pass を導入することで解決した（図 5.3）。すなわち、親関数の入口に戻るためには、scan pass と re-exec. pass の 2 パスが必要になる。

scan pass では、デバッガの持つスタック検査コマンド[1]を用いてスタックの深さを正確に知る。従って、scan pass では通常のデバッギの実行時間に加えて、デバッガによるスタック検査の時間がかかってしまう。

[1]GDB では `where`。

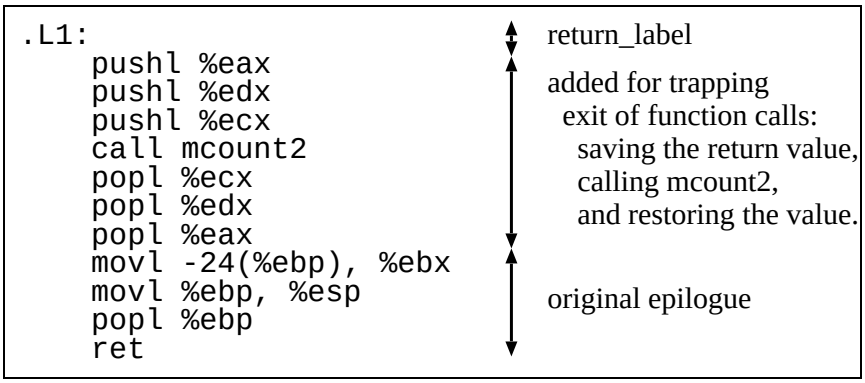

図 5.4: 関数呼び出しの出口のトラップ

5.1.3 新手法

2 パスを要する問題は、関数呼び出しの出口を捕捉して、図 5.3 の通常実行中においても呼び出し深さを正確に保持することができれば解決する [MT00]。関数呼び出しの入口部分については、従来手法同様にプロファイル用関数の機構を流用し、関数呼び出しの出口の仕組みはアセンブリコードレベルで実装した[2]。これにより、親関数の入口へ戻るのを 1 パスにすることができた。

この場合は関数呼び出しの出口の処理が `return` 文の場合も兼ねているので、返り値が保存されている可能性のあるレジスタを全て保存してから、タイムスタンプ更新のための関数を呼び出し[3]、再びレジスタの内容を復元する、という手順を行っている（図 5.4）。

新手法の処理時間を、以下の条件で計測した。デバッギとして sed-1.18 を用い、10816 行（1379312 バイト）のテキストファイル[4]に対して `s/make/MAKE/g` を行って `/dev/null` へ出力した場合について、126471 回目の関数呼び出しで停止させ、その呼び出しの入口への逆実行に要する時間を計測した[5]。計測に用いた計算機は i486DX2 50MHz、メモリ 24MB、Linux-2.0.35 という構成で、時間の計測には GNU time-1.7 での elapsed time を用いた。以下の 2 種類の処理時間を、従来手法と新手法について調べた。

[2] 第 4.1 節で述べた通り、このレベルで実装するのが最も容易である。

[3] ここでは、前述のマクロ `INC_TS` に相当する関数 `mcount2` を用意して、これを呼び出すような実装になっている。

[4] Linux カーネルを make したときのログファイルを連結したもの。

[5] 親関数の入口への逆実行ではないが、処理のオーバヘッドを知るには十分である。

表 5.1: 新手法と従来手法による sed の処理時間 [MT00]

	old method	new method	optimized
normal exec.	4.816	4.854	3.088
directrun	14.126	14.114	—
rev-myself	28.574	21.136	—
without scan pass	21.194	—	—

(sec.)

通常実行 -g オプションと -pg オプションでコンパイルしたデバッギを通常実行[6]。この場合だけ、特に参考のために、-O3 オプションでコンパイルしたデバッギの実行時間も計測した。

directrun 上記のデバッギを、逆実行を実装した Perl スクリプト（以下、ドライバプログラムと呼ぶ）で 1 パスだけ実行。このスクリプトは GDB を起動し、これを介してデバッギを制御するので、この時間には Perl と GDB の起動時間も含まれる。

rev-myself 上記のデバッギで、ドライバプログラムによって自分自身の関数呼び出しの入口へ逆実行した場合の実行時間。

without scan pass 従来手法に関してのみ、scan pass を行わなかった場合の処理時間も計測した[7]。

計測の結果を表 5.1に示す。関数呼び出しの出口を捕捉するようになったことで、通常実行時のデバッギの実行速度は従来手法よりも低下しているとは言え、微々たるものである。

一方、表 5.1より得られた実行時間の内訳は図 5.5のようである。Perl と GDB の起動に要した時間を除いた実行時間はおよそ 0.65 倍になっており、逆実行速度は 3 パスから 2 パスになった分、減少し、新手法の効果を示している。

[6] プロファイル用関数の仕組みを流用するので、-pg が必要になる。新手法用のデバッギは、これらオプションを指定して得られたアセンブリコードをさらに変換して、実行コードを得る。

[7] この場合に限っては最初から目標点のタイムスタンプが分かっているので、scan pass を不要にしても目標点に制御を移動できる。

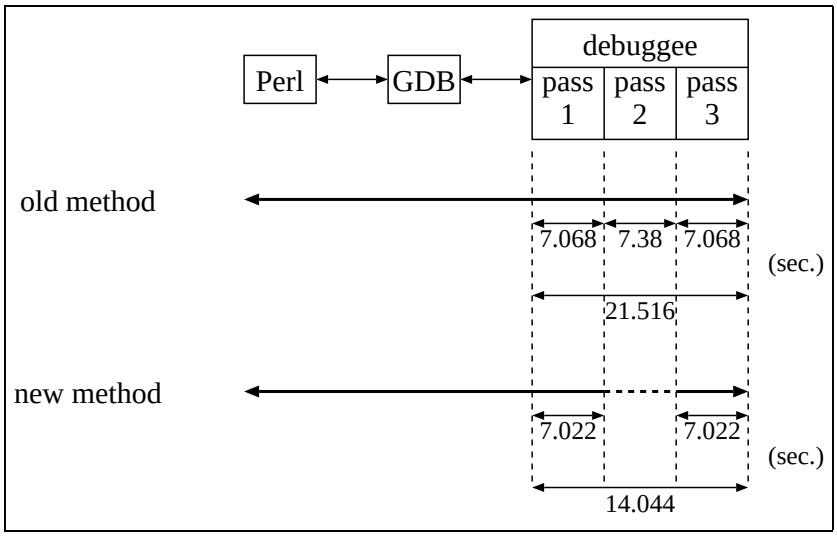

図 5.5: 各手法の処理時間の内訳

5.1.4　本論文での位置付け

ここで述べた手法では、タイムスタンプを関数呼び出しの入口でしか更新しない。これは前章で述べた現行のタイムスタンプの更新間隔を、より疎らにしたものであると考えられ、限られた点しか指定できない動的ブレークポイントであるとも言える。また、実装はアセンブリコードレベルを採用し、実装の容易さのためにポータビリティを犠牲にした。

この新手法は本研究の予備的な研究であり、この後、実行履歴中の任意の 1 点を特定するためにタイムスタンプの更新間隔をより細かくすべく、タイムスタンプ更新タイミングを 4 つに増やした[8]。

一方で、前章で述べた現行の position の実装では、この関数単位逆実行をそのまま実現することはできない。現在は、 `INC_TS` はタイムスタンプの更新だけで、図 5.2のようなスタックのシミュレートを行っていないからである。現行の実装で関数単位逆実行を実現するには、関数呼び出しの際に実行されるタイムスタンプ更新機構に、同様のスタックのシミュレートを組み込めば良い。

5.2　実行点の印づけ

[8]関数呼び出しの入口、出口（`return` 文の場合を含む）、ループ、`goto`。

通常のデバッグにおいてバグの原因である場所が曖昧にしか分かっていないとき、その場所の手前に（静的）ブレークポイントを設定して制御を移動し、ステップ実行を繰り返し行う、という方法で実行点を動かす。大規模なプログラムの中をこの方法で移動していると、うっかりコマンドを間違って問題の場所を通り過ぎてしまうことがある。このようなときには、そこへ至る経路を覚えておくか書き留めるかしておき、デバッギを先頭から再実行してその経路を手動で追いかけなくてはならない。

こうした面倒を避けるためには、実行点、すなわち position に印づけをすることが有効である。デバッギの振舞が「ここまでは正しい」というときに動的ブレークポイントを用いて印をつけ、後になってやり直したいようなことがあったらその position まで簡単に戻ってくることができる。

さらに、印をつけたいくつもの position を区別するのに、単なる ID 番号ではなく、自分でコメント文を書くことによって識別をより簡単にすることもできる。コメントとしては、例えばこんなものがあるだろう。「今、右ブレースを読み込んだので、複合文の処理をするパーザアクションに制御が移るに違いない。」

5.3 逆向きウォッチポイント

メモリを動的に確保するようなプログラムを考える。`malloc` のようなアロケータ関数でメモリを確保するものの、プログラマは誤って、確保した領域よりも大きなデータを書き込んでしまい、`malloc` や `free` が使うような管理領域の情報を破壊してしまう。しかしその破壊操作は直ちにはエラーを引き起こさず、バグの影響はずっと後になって顕在化する。配列に、その確保した領域を越えてデータを書き込んでしまうときも同様で、プログラマは後になって代入した覚えのない変数の値がおかしくなっていることに気づく。

このような不正な書き込みを捕捉するために、デバッガにはデータアクセスブレークポイントという仕組みが備わっており、GDB ではウォッチポイントと呼ばれている。おかしな値になってしまった変数に対してウォッチポイントを設定すると、その変数に対応するメモリ領域へ書き込みが発生する度にトラップが起こり、その内容が変更されたことをプログラマに通知する。

一般には変数は何度も書き換えられるものであり、その度にトラップが発生して、プログラマはそれぞれの書き込みで何が起こったかを検査しなくてはならない。しか

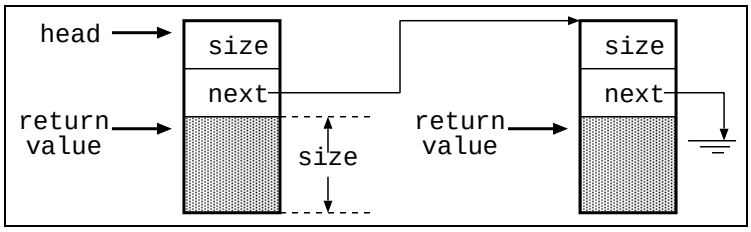

図 5.6: **simple_alloc** による領域管理

しながら、多くの場合、ある問題を引き起こした書き込みというのはそのバグの影響が現れた時点から時間的に一番近いものが原因になっていて、実際に我々はそれを探すことを手作業で行っている。従って、ウォッチポイントによる直前のトラップまで自動的に制御を移す仕組みがあれば、プログラマの作業がかなり軽減されることになる。この機構を「逆向きウォッチポイント」と呼ぶことにする。

5.3.1 利用例

C における **malloc** のような、メモリ確保をするプログラムを考える。関数 **simple_alloc** は引数で指定された大きさの領域を確保し、確保した領域のアドレスを返す。 **simple_alloc** は領域を線形リストで管理し、新しい割り当ては first-fit に基づいて行う。管理のために、割り当てられた領域にはヘッダ情報として、 (1) その領域の大きさと (2) 次に割り当てられた領域のヘッダへのポインタが格納されている (図 5.6) 。関数 **simple_free** は、 **simple_alloc** で確保した領域が不要になったときに、再利用できるようにヒープに返すのに用いる。このプログラムの **main** 関数を図 5.7に示す。

このプログラムは、最初に行 (2) で 4 バイトの領域を **area1** に確保する。次に、別の 4 バイトを **area2** に確保し (4) 、それを解放するという操作を 253 回繰り返した後で、再度 **area2** に確保をする (6) 。そして関数 **string_copy** を用いて **area1** に文字列をコピー (7) するときに、 **area2** の管理領域を破壊する。最後に **area3** に領域を確保して終了する。 **area3** が確保された後のヒープの状態は、図 5.8のようになっているはずであるが、このプログラムは segmentation fault を引き起こす。早速、これをデバッグしてみる。

```
    int main(void){
      int i;

(1)   init_heap();
(2)   area1 = simple_alloc(4);
(3)   for(i = 0; i < 253; i++){
(4)     area2 = simple_alloc(4);
(5)     simple_free(area2);
      }
(6)   area2 = simple_alloc(4);
(7)   string_copy(area1, "abcdefghijk");
(8)   area3 = simple_alloc(4);
(9)   return(0);
    }
```

図 5.7: メモリ確保プログラム

バグ発覚場所の特定

このプログラムをデバッガを用いて動かすと、バグは図 5.7の行（8）の **simple_alloc** を呼び出したところでエラーとして発覚していることが分かる。これは **area3** に領域を確保しようとしているところである。 **simple_alloc** は first-fit なので、 **area3** は **area2** のすぐ後ろに割り当てられるべきである。 **simple_alloc** は空き領域を探すために先頭（**head**）から **next** をたどっていって、 **area2** の **next** にアクセスしたはずである。そこで **area2** の管理領域をデバッガで調べてみると以下のように表示された。

```
(gdb) print head->next->size
$2 = 1751606885
(gdb) print head->next->next
$3 = (struct cell *) 0x6b6a69
```

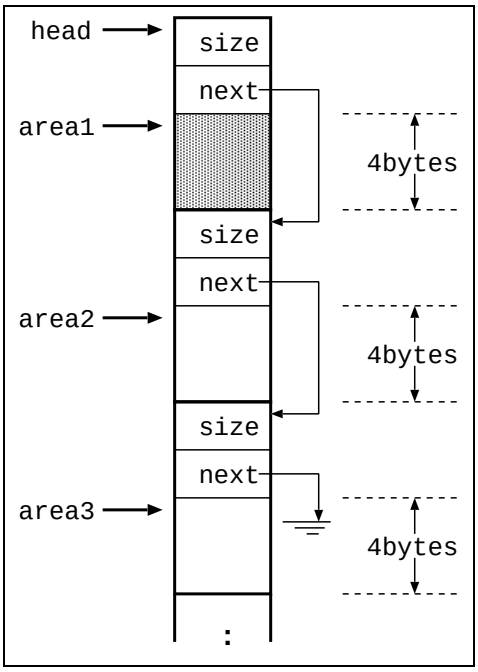

図 5.8: 本来のヒープの構造

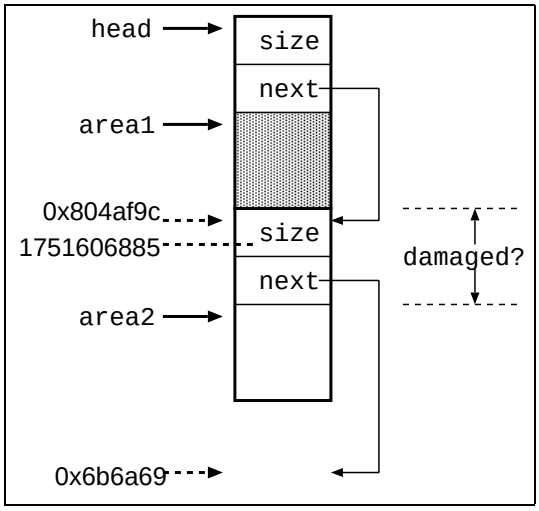

図 5.9: 破壊されたヒープ

area2 は 4 バイトのはずなのにかなり大きなものになっている。しかも線形リストの末尾のはずが、その次のセルがひどく遠いところにあることになっている（図 5.9）。ここに至って、何らかの操作によって **area2** の管理領域が破壊されたと推測できる。

逆向きウォッチポイントによるデバッグ

そこで、注目する領域に最後に書き込んだ操作を知るために逆向きウォッチポイントを用いる。`area2` の管理領域（0x804af9c）に対して逆向きウォッチポイントを行うと、以下のような出力が得られた。

```
Old value = 6776421
New value = 1751606885
string_copy (loc=0x804af98 "abcdefgh",
       string=0x8049758 "abcdefghijk")
    at allocator.c:145
145         i++;
```

これで、最後の操作が行（7）で呼び出された関数 `string_copy` 内であり、4 バイトしかない領域に大き過ぎるデータを書き込んだのが原因だと分かった。

通常のデバッグ

もし逆向きウォッチポイントがなかったら、このデバッグはやっかいなものになる。実行の最初に通常のウォッチポイントを設定してからデバッグを始めると、対象領域はループを回る度に繰り返し書き換えられるので、デバッギは何度も停止してしまう。そして、止まる度に表示された値が正当なものであるかを調べ、デバッガに継続を指示しなくてはならない。一般にはこの停止が何回起こるかは分からないので、プログラマは大量の時間と労力を浪費することになる。

5.3.2 逆向きウォッチポイントの実装

逆向きウォッチポイントは前章で述べた動的ブレークポイントと既存のデバッガを用いれば容易に実装することができる。図 5.10 にそのアルゴリズムを示す。

1. 逆向きウォッチポイントが指示された position （図中の **S**）に動的ブレークポイントを設定する。

2. pass 1 : デバッギを再実行し、ウォッチポイントによるトラップで停止したときは、その position に印づけをするために、動的ブレークポイントを設定する

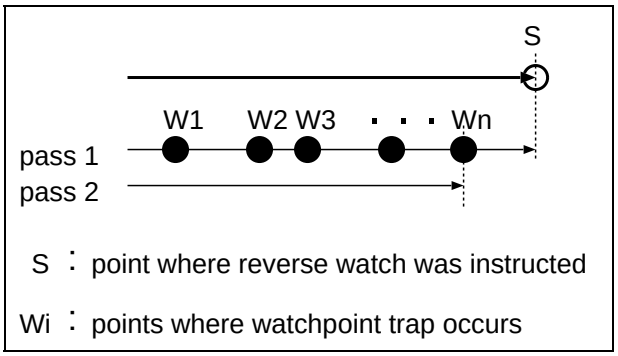

図 5.10: 逆向きウォッチポイントのアルゴリズム

ために必要な情報である (file:line, timestamp) を取得する。図中の **W1** から **Wn** がこれにあたる。それから実行を続行する。

3. 制御点が逆向きウォッチポイントを指定した position に到達したら（動的ブレークポイントを設定したのでこれを知ることができる）、

4. pass 2： もう一度デバッギを再実行し、ウォッチポイントによる最後のトラップが発生した場所（図中の **Wn**）に、新たに動的ブレークポイントを設定してから実行を続行する。

5. 目標の position である **Wn** で停止する。

逆向きウォッチポイントはデバッガとのインタラクションが必要になるので、新しいデバッガは Expect モジュールを用いた Perl スクリプトで記述し、GDB のラッパーとして実装した。GDB は GDB-5.0 を用いた。

5.3.3 逆向きウォッチポイントの処理時間

逆向きウォッチポイントの処理にかかる時間を計測した。デバッギには図 5.7に示したものを用いた。この処理時間には以下のものが含まれる。

- Perl と GDB の起動時間。
- pass 0： segmentation fault で停止する、通常実行パス。

表 5.2: 逆向きウォッチポイントの処理時間

trap of watchpoint (times)	elapsed time (sec.)
511	6.56
1011	11.94
1511	17.58
2011	23.46

- pass 1： 逆向きウォッチポイントの pass 1。デバッギを先頭から再実行し、ウォッチポイントで停止する度に、デバッガを用いて現在の position 情報である (file:line, timestamp) を取得、保存する。

- pass 2： 逆向きウォッチポイントの pass 2。 pass 1 で取得したリストの最後の position まで、動的ブレークポイントで用いて先頭から再び再実行する。

逆向きウォッチポイントの処理時間は pass 1 におけるウォッチポイントの停止回数に依存するので、これを 511 回から 2011 回まで変えて計測を行った。計測に用いた計算機は Celeron 400MHz、 Linux-2.0.36 であり、 GNU time-1.7 で elapsed time を計った。結果を表 5.2に示す。

デバッギそのものの実行時間は `time` コマンドで計測できないほど微々たるものであったので、表 5.2の結果は逆向きウォッチポイントの処理時間そのものであるとみなしても差し支えない。処理時間はウォッチポイントの停止回数にほぼ比例しており、停止回数はデバッギの総実行時間にほぼ比例すると考えられるので、実用上問題のない実行時間であると考える。

逆向きウォッチポイントは、特定の条件を満たす position のリストを取得して、そのうちの最後の position に制御を移動している、とみなすこともできる。逆向きウォッチポイントでは、その条件として対象の変数へのウォッチポイントを用いたが、条件付きブレークポイントのような他の条件も利用できるし、制御の移動先として最後の点ではなく中間点を使うこともできる。次節ではその例を挙げる。

5.4 タイムスタンプを用いた実行履歴中の 2 分探索

ここまでは、これまでプログラマが手動で行ってきた手続きを自動化するというものであった。ここでは、手動では到底できないような手続きを実行する例について述べる。

逆向きウォッチポイントは特定の変数に対して設定したので、その変数を監視してバグの原因を自動的に探すことができた。これに対して本節では、ある条件が成立しなくなるときの position を自動的に探してバグ特定を補助する仕組みとして、タイムスタンプを用いた実行履歴中の 2 分探索（以下、単に 2 分法と呼ぶ）を提案する。

タイムスタンプというのはステップ実行カウンタをもっと疎らにしたもので、その値はプログラム実行履歴中における大まかな位置を表していると考えられる。例えば、タイムスタンプ値が 1 で始まり 10000 で終わるようなプログラムを考えると、その値が 5000 になる場所は実行履歴において概ね半分の位置とみなせるということである。そこで、タイムスタンプ値をキーとする 2 分法を用いて実行履歴中を移動することで、ある条件が不成立になるときの position を見つけられる。

条件の例として、双方向リストを構成するプログラムがあるとすると、そのリストの一貫性を調べるためのある種のアサーションをコード中に用意しておけば、それが最初に偽になるときの position を知ることができる。例えば、リストの先頭から順方向リンクをたどって末尾まで行き、逆方向リンクをたどって先頭まで戻ってくることができる、といったアサーションが考えられるであろう。

双方向リストのデータ構造の一貫性を調べるアサーションであれば、デバッギ実行中のその真偽は、通常、真から偽へ 1 回変化するだけであり、偽（異常な状態）になってから復旧することはまずあり得ない。しかし、アサーションやデバッギによっては、デバッギ実行中に真になったり偽になったりを繰り返す場合もあり得る。そのようなときでも、この手法を用いれば、いくつかある「真から偽への変化」のうちの 1 つは見つけることができる。

5.4.1　2 分法のアルゴリズム

2 分法を実行するプログラムは、以下のようにしてデバッギを制御すれば良い。

1. タイムスタンプの値が左端と右端の値の中間値になる position へ、制御を移動する。最初は、左端の点は実行履歴の先頭に等しく、右端の点は履歴の末尾に等しい。

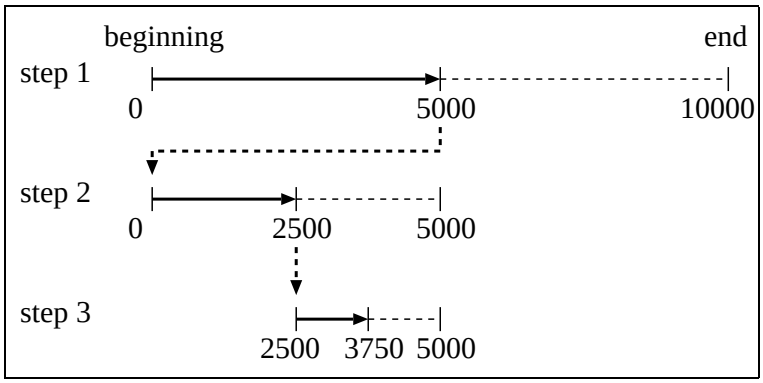

図 5.11: 2 分法の例

2. 条件を評価し、

- 真であれば、現在点と右端の中間点へ移動し、左端の点を新しい現在点に変更して対象領域を半分にする。
- 偽であれば、左端と現在点の中間点へ逆実行を用いて戻り、右端の点を新しい現在点に変更して対象領域を半分にする。

3. 以上を繰り返す。

この例を図 5.11に示す。タイムスタンプ値が 10000 まで増えるプログラムを扱うとする。 step 1 では、 1 と 10000 の中間値である 5000 まで進む。条件を評価して偽であったとすると、 step 2 に移り、デバッギを再実行して左端（タイムスタンプ値が 1）と現在点（5000）の中間点まで戻る。タイムスタンプ値が 2500 の場所で条件を評価して真になったとすると、 step 3 で現在点の右端の中間点（タイムスタンプ値が 3750）へと進む。この手続きを繰り返すと、指定の条件が最初に偽になる position が見つけられる。

5.4.2　2 分法と `assert` との比較

条件が不成立になる状況とは、プログラムが **`assert`** ライブラリ関数で異常終了する場合と同等であると考えられるので、 2 つの利点を除いては、 2 分法は **`assert`** を用いて実行を停止させることに等しい。 2 分法が優れる点のうちの 1 つは、 **`assert`** は必要になるかも知れない全ての場所（location）に手動で挿入しなければならない、という点である。 2 分法では手動で挿入する必要はない。もう 1 点は、挿入され

た `assert` はその呼び出しが起こる度に条件を評価してしまう、という点である。2 分法を用いれば、タイムスタンプの最大値（実行履歴末尾でのタイムスタンプ値）を n とすると評価の最大回数は $\log n$ となるので、条件が複雑なものであれば、条件の評価に要する時間は 2 分法の方が `assert` よりも短くなる。

2 分法の処理に要する時間の大半は、条件の評価とデバッギ自身の実行時間である。条件評価は高々 $\log n$ 回であり、デバッギ自身の実行時間の合計も 1 回の実行時間の高々 $\log n$ 倍で済む。タイムスタンプとして 32 ビットの符号なし整数を用いれば、条件評価は 32 回以内、実行時間は 32 倍以内と見積もることができ、実現可能な範囲にあるものと考える。

第 6 章

各種の議論

本章では、第2章で挙げた各研究との比較を行う。

6.1 履歴保存型逆実行

第2.1.1節で挙げた各研究は、全て何らかの形で実行状態を保存することで逆実行を実現しており、本研究の手法とは根本から異なる。それぞれの差異は保存すべき実行状態の量をいかにして減らすかという点である。

Lieberman ら [LF97] と Moher[Moh88] は、それぞれ逆実行可能なデバッグ環境の提案と実行状態の視覚化を目的としており、実行速度の効率化には重点が置かれていないため、小規模なプログラムのみを対象とした教育目的のような用途に限定されてしまう。

Feldman ら [FB88] の仮想記憶のページ単位で保存する方法では、表 6.1のような結果が示されている。これは、`make` と C コンパイラと `sort` の各アプリケーションで状態保存間隔（checkpointing interval）を変化させたときの実行速度の低下を表している。十分な解像度を得るにはより細かい間隔で状態を保存する必要があるが、最もオーバヘッドが少ない `sort` の場合でも、保存間隔が 0.1 秒のときは実行時間が 2 倍以上になっている。また、近年の高速なプロセッサにおいても同程度のオーバヘッドで済むとは考えにくい。

Agrawal ら [ADS91] は、動的プログラムスライシングが有効に機能する言語を対象にして実装を行っているが、C のようにポインタがある言語では効果的ではない。

Biswas ら [BM99] の inverse statement は、`a /= b;` のようなステートメントで元

表 6.1: IGOR における保存量の低下 [FB88]

	Make		Ccom		Sort	
Checkpointing Interval (seconds)	Execution Time (seconds)	Overhead	Execution Time (seconds)	Overhead	Execution Time (seconds)	Overhead
none	4.7	–	12.1	–	22.2	–
1.0	8.0	70%	24.3	101%	31.7	43%
0.3	11.3	140%	34.3	183%	37.3	68%
0.1	22.7	380%	46.5	280%	47.5	114%

の値を完全に正しく復元する逆演算を定義できないので、変数の値を保存せざるを得ない。また、ソースコードがないライブラリ関数の呼び出しが起こる時は、どの変数の値が変更されるかが分からないので、全ての変数を保存するとしている。また、条件分岐やループなどで制御が不連続にジャンプするときには、行番号を履歴として保存する。文献中には保存される履歴の量に関する記述はないが、実用的なほどに小さくなるとは考えられない。

これらの方法は履歴保存量をいくばくかは削減しているがいずれも決定的な手法ではなく、プログラムの大規模化に伴って実行履歴は巨大にならざるを得ない。また、いずれも専用の処理系やデバッガを実装しているが、これはかなり骨の折れる作業で、プログラミング言語のフルセットの機能を実装するのは困難であり、またポータブルな実装もできないので現実的な手法とは言えない。

6.2 イベント駆動型実行制御

Ducassé[Duc98] の方法は、ブレークポイントの条件を加えていくことによって該当する position を絞り込んでいると考えることもできる。図 2.3で例に挙げたような複雑な条件を正しく指定することができれば、本研究で提案する実行点の特定と似通った機能を持つことになる。しかしながら、どんなに厳しい条件を付加しても position を区別できるとは限らないので、完全に 1 点を特定することはできない。

この方法はプログラマが手動で実行履歴中を調べて歩くのには向いている手法だが、条件の指定が難しく、プログラマにとってはデバッグ作業がより複雑なものになってしまう。本研究ではタイムスタンプを利用したより単純な仕組みを用いること

表 6.2: Boothe の方法での順方向コマンドのオーバヘッド [Boo00]

command	slowdown
step	1.95
continue	1.95
next	2.48
finish	2.33
until	2.34

で、計算機による自動化に向いた基礎を提案した。

6.3 タイムスタンプの更新間隔

本研究では、第3.3節で述べたような規則に従って、目標点を特定するためのカウンタ（タイムスタンプ）を更新することとしたが、Boothe[Boo00] が行ったようなステートメント単位でのタイムスタンプ更新も可能である。

彼は完全な reversible debugger を目指したので、直前のステートメントに容易に「逆向きステップ」するためにこれを必要としていたが、我々の目的はプログラム実行履歴中の 1 点を特定することなので、ステップ実行カウンタはオーバスペックである。

また、ステップ実行カウンタはデバッギの完全な実行履歴をカウンタの形で保持していると考えられ、動的な情報だけで構成されている。本研究で提案するところの position は動的な情報であるカウンタの値だけを用いるのではなく、静的な情報である location、(file:line) と組み合わせることにより、低い更新頻度を実現したと言える。実際、Boothe の方法では、最もオーバヘッドの少ない順方向のステップ実行でも 2 倍近くの時間を要している（表 6.2）。これに対して本研究では、表 4.1に示したように 1.4 倍前後で済んでいる。

第 7 章

結論

7.1 結論

デバッグ作業の自動化を進めていくためには、プログラムが他のプログラムの実行を制御できる必要があり、プログラマが対話的に扱うことを前提として作られているツール類と、それらが現在持っている仕組みだけでは不十分である。

本研究では、プログラムの実行履歴において一貫性が保たれる、動的な位置（position）を提案した。これを実現するためにタイムスタンプと呼ぶカウンタを導入し、静的に unique な情報である (file:line) と組み合わせて、 (file:line, timestamp) という組で position を表した。これを代表的なコンパイラである GCC に実装する方法を述べ、デバッギの速度低下も実用上問題とならない程度であることを示した。 position を用いて実行履歴中に設定するブレークポイントとして、動的ブレークポイントを提案し、 GDB を用いて実装した。

そして、この position を用いて典型的なデバッグ作業を自動化できることを示し、逆向きウォッチポイントについて実行時間を計測、実用的な時間で動作することを確認した。

7.2 今後の展望

本研究では、 position を応用して自動化されるデバッグ手法として、以下の 3 つを

挙げた。

- 再実行による逆実行
- 逆向きウォッチポイント
- タイムスタンプを用いた実行履歴中の 2 分探索

これら以外にも、他の手法を確立し、実装していくことができる。そのためにはまず、バグの分類を行うことが必要である。バグのような「誤り」に関する情報は管理されにくいものであり、また、公開されることも少ない。どのようなバグが生じているかという情報を収集し、そのようなときにはどのような手法が適用されているか、あるいは position を用いることで手動では不可能な新しい手法を実現することができないか、を考えることになる。そして、これら手法とともに、本研究で提案したデバッグ手法の有効性を実証的に評価する必要がある。このためにも、バグ情報の収集は不可欠である。

また、今回は再現性のある C プログラムに限定したが、再現性のないプログラムへの対処も重要である。これを完全に扱うには、やはり定期的に状態を保存するしかないが、対象プログラムに条件を付けて適切に分類すれば、完全な保存に頼らなくとも再現性を持たせることが可能になる。例えば、非同期の入出力を行うプログラムは再現性を持たないが、入出力が発生するタイミングとその内容を保持しておけば再現性を持たせることができる。マルチスレッドの並行プログラムであっても、スレッドの切替えタイミングと実行順を保持することができれば再現可能になる。

再実行に要する時間の短縮手法として、実行状態の保存を採り入れることもできる。 Boothe[Boo00] は再実行におけるオーバヘッドを緩和するために、定期的に実行状態を保存するチェックポインティングを行っている。ゲームプログラミングにおける木の探索など、非常に時間がかかるプログラムを対象とする場合には重要な機能となる。

謝辞

本研究は東京大学大学院工学系研究科機械情報工学専攻算法設計工学研究室において、同研究科情報工学専攻の修士論文として行われました。

指導教官である寺田実助教授には、研究の方針やアイディアについて数々の御助言を賜わりました。御礼申し上げます。

Nigel Ward 助教授には、研究室輪講等で、寺田助教授とは異なる視点から貴重な御指摘、御批評をいただき、大変有難く、感謝致します。

東京大学情報基盤センター情報メディア教育研究部門の田中哲郎助教授には、UtiLisp/C のバグレポート情報を供出していただきました。心より御礼申し上げます。

永井おりが文部科学技官には、研究における事務作業等に御助力いただき、御礼申し上げます。

本研究室博士課程卒業生の立山義祐氏（現、通信・放送機構ぎふ MVL リサーチセンター）、塚原渉氏（現、株式会社日立製作所中央研究所）、小林広和氏（現、株式会社日立製作所システム開発研究所）には、研究室輪講や勉強会などで御指導、御意見をいただき、感謝致します。

同修士課程学生の首藤達生君、樋口直志君、須山哲宏君、同学部学生の副田俊介君、西本圭介君には、研究を進めていく上で率直な御意見と惜しみない御協力をいただき、感謝しています。特に須山君には御礼の言葉もありません。

最後に、ここにお名前を挙げきれなかった研究室の皆様と、情報工学輪講や情報処理学会プログラミング研究会など、研究発表の場で御助言を下さった皆様に、心より御礼申し上げます。

参考文献

[ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An Execution-Backtracking Approach to Debugging. *IEEE Software*, Vol. 8, No. 3, pp. 21–26, 1991.

[BM99] Bitan Biswas and R. Mall. Reverse Execution of Programs. *SIGPLAN Notices*, Vol. 34, No. 4, pp. 61–69, 1999.

[Boo00] Bob Boothe. Efficient Algorithms for Bidirectional Debugging. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pp. 299–310, 2000.

[Duc98] Mireille Ducassé. Coca: A Debugger for C Based on Fine Grained Control Flow and Data Events. In *Rapport de recherche INRIA RR-3489*, 1998.

[FB88] Stuart I. Feldman and Channing B. Brown. IGOR: A System for Program Debugging via Reversible Execution. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and distributed debugging*, pp. 112–123, 1988.

[Inc92] D. C. Ince. *An Introduction to Discrete Mathematics, Formal System Specification, and Z, Second Edition.* Oxford University Press Inc., 1992.

[Ish86] 石井康雄（編）. ソフトウェアの製造, 日科技連ソフトウェア品質管理シリーズ, 第3巻. 日科技連, 1986.

[LF97] Henry Lieberman and Christopher Fry. ZStep 95: A Reversible, Animated Source Code Stepper. In Blaine Price John Stasko, John Domingue and Marc Brown, editors, *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1997.

[Mey91] Bertrand Meyer. *Introduction to the Theory of Programming Languages.* Prentice Hall, Inc., 1991. (二木監訳, 酒匂訳：プログラミング言語理論への招待, アスキー出版局, (1995)).

[Moh88] Thomas G. Moher. PROVIDE: A Process Visualization and Debugging Environment. *IEEE Transactions on software engineering*, Vol. 14, No. 6, pp. 849–857, 1988.

[MT00] 丸山一貴, 寺田実. 関数単位疑似逆実行の高速化. 情報処理学会論文誌 プログラミング, Vol. 41, No. SIG9(PRO8), pp. 1–7, 2000.

[Mye79] Glenford J. Myers. *The art of software testing.* John Wiley & Sons, Inc., 1979. (長尾監訳：ソフトウェア・テストの技法, 近代科学社, (1980)).

[Ros97] Jonathan B. Rosenberg. *How Debuggers Work.* John Wiley & Sons, Inc., 1997. (吉川訳：デバッガの理論と実装, アスキー出版局, (1998)).

[SPS+00] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB – The GNU Source-Level Debugger – Eighth Edition*, 2000. (included in GDB source tree).

[Sta99] Richard M. Stallman. *Using and Porting the GNU Compiler Collection.* Free Software Foundation, 1999. (included in GCC source tree).

[Ter00] 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.

付録 A

GCC への修正

ここでは、タイムスタンプ機構の GCC への実装について、GCC のソースコード中に挿入した実際のコードを示す。

A.1 タイムスタンプ更新コード用初期化手続き

タイムスタンプ更新マクロ `INC_TS` を挿入する前に、元のデバッギでは宣言されていなかった 3 つの識別子、`timestamp`、`ref`、`inc_ts_stop` の宣言を環境に追加しなくてはならない。そのための関数 `init_for_inc_ts` を図 A.1に示す。この関数は、構文解析器である `yyparse` によって、各ソースファイルにつき 1 回だけ呼び出される。関数 `start_decl` を呼び出して、それら 3 つの識別子を順に宣言している。

A.2 タイムスタンプ更新コード挿入手続き

タイムスタンプ更新マクロ `INC_TS` を RTL として挿入するために GCC に追加した関数 `emit_inc_ts` を図 A.2に示す。第 4 章で述べた GCC の各アクションの適切な位置で、`-pg2` オプションが指定されているときにだけ呼び出される。

最初の `c_expand_start_cond` が、`if` 文の条件の評価をして、それによって then パートを実行するか否かを決定する。次の `expand_expr_stmt` は、then パートである `inc_ts_stop` の呼び出しを展開している。最後の `c_expand_end_cond` は、条件を評価した結果が偽であったときのジャンプ先ラベルを設定する。

```
void
init_for_inc_ts ()
{
  tree d = start_decl(
             get_identifier("timestamp"),
             chainon(NULL_TREE,
                     tree_cons(
                       NULL_TREE,
                       ridpointers[RID_INT],
                       tree_cons(NULL_TREE,
                                 ridpointers[RID_EXTERN],
                                 NULL_TREE))),
             0, 0, 0);
  finish_decl(d, NULL_TREE, 0);
  d = start_decl(
        get_identifier("ref"),
        chainon(NULL_TREE,
                tree_cons(
                  NULL_TREE,
                  ridpointers[RID_INT],
                  tree_cons(NULL_TREE,
                            ridpointers[RID_EXTERN],
                            NULL_TREE))),
        0, 0, 0);
  finish_decl(d, NULL_TREE, 0);
  d = start_decl(
        build_nt(CALL_EXPR,
                 get_identifier("inc_ts_stop"),
                 get_parm_info(1),
                 NULL_TREE),
        tree_cons(NULL_TREE, ridpointers[RID_VOID], 0),
        0, 0, 0);
  finish_decl(d, NULL_TREE, 0);
}
```

図 A.1: 関数 init_for_inc_ts

```
void
emit_inc_ts ()
{
  c_expand_start_cond(
    truthvalue_conversion(
      build_compound_expr(
        build_tree_list(
          NULL_TREE,
          parser_build_binary_op(
            EQ_EXPR,
            build_unary_op(PREINCREMENT_EXPR,
                           lookup_name(
                             get_identifier("timestamp")),
                           0),
            lookup_name(get_identifier("ref"))))))),
    0, compstmt_count);

  expand_expr_stmt(
    build_compound_expr(
      build_tree_list(NULL_TREE,
        build_function_call(lookup_name(
                              get_identifier("inc_ts_stop")),
                            NULL_TREE))));

  c_expand_end_cond();
}
```

図 A.2: 関数 emit_inc_ts