平成 12 年度夏学期 第 1 回情報工学輪講資料

Reverse Watchpoint

寺田研究室　修士 2 年　丸山 一貴*

05-15-2000

[出典]

Kazutaka Maruyama and Minoru Terada,
"Reverse Watchpoint",
AADEBUG'2000
– Fourth International Workshop on Automated Debugging –,
Aug. 2000 (submitted).

要旨

バグの原因を特定することは、バグの影響がその原因よりもずっと後で発覚する場合、特に困難である。そのようなバグを調査するのに有効な手段として、デバッガの持つウォッチポイント（データ・アクセス・ブレークポイント）の仕組みがあるが、プログラムの実行を順序通りに追跡するため、プログラマはバグの原因ではない場合にも、デバッガからの報告を吟味しなければならない。

しかし多くの場合、その報告のうちの最後のもの、すなわちバグの影響が出た場所の直前の報告がバグの原因となっている。そこで、

1. デバッガからの報告の履歴を自動的に取得し、
2. その最後の場所にプログラムの制御を移す

ための仕組みである、"reverse watchpoint" を提案する。これは、ソースコードの変換とデバッガに多少の拡張を施すことで実現でき、現実的な実行時間で動作することを示す。

1　はじめに

逐次型言語におけるプログラムの実行は、メモリ領域の書き換えの系列と考えられる。高級言語では、メモリ領域はコンパイラによって管理され変数と対応づけられているので、ある領域の値を変更することは対応する変数の値を変更することに他ならない。しかしながら、C におけるポインタのように任意の領域に自由にアクセスできると、対応する変数を介さなくてもその領域にアクセスできてしまう。これは aliasing と呼ばれる。aliasing が起こる場合、ある変数の値がおかしくなったときに、その変数への代入文だけを調べるだけでは原因を特定できないことがあり得る。

そのため、デバッガの中には "watchpoint"（Data Access Breakpoint）と呼ばれる仕組みを持つものがある。これは、特定のメモリ領域を変更するオペレーションを監視するものであり、デバッガはそのようなオペレーションを感知すると、デバッグ対象プログラム（以下、デバッギと呼ぶ）の実行を停止して、プログラマに制御を返す [1]。この機能があれば、おかしな値になった変数の領域に watchpoint を設定しておくことで、その領域に対する全てのオペレーションを捕捉することができる。しかし、ある領域を変更するオペレーションは一般には何度も起こるものであり、デバッギはその度に停止してしまうので、その最後のオペレーションを見つけるのは大変面倒である。

そこで、手続き型・逐次型で再現性のあるプログラムに対して、その最後のオペレーションが起こった点に制御を移す、"reverse watchpoint" を提案する。以下ではまず、デバッガの持つ breakpoint と watchpoint について簡単に説明し、reverse watchpoint の概要とそれが有効な例を示し、その実装を説明してオーバーヘッドを示す。

*kazutaka@sanpo.t.u-tokyo.ac.jp

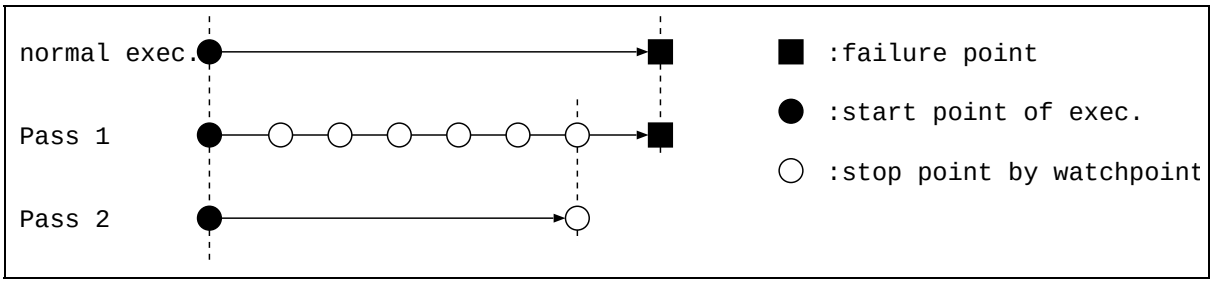

図 1: watchpoint によるデバッグ手順

2 breakpoint と watchpoint

この節では、デバッガの breakpoint と watchpoint という仕組みについて説明する。

2.1 breakpoint

デバッガでデバッギの実行をトレースする場合、大きく分けて 2 つの手段がある：

ステップ実行 ソースコードの 1 行を単位として、順に実行する

breakpoint ソースコードの「ある行」が実行されるときに、停止する

ステップ実行はデバッギの振舞を細かく追うことができるが、逆に細か過ぎてしまって不要な部分をうまく省略することが難しく、手間も時間もかかってしまう。一方、 breakpoint は「ここが怪しい」とあたりをつけたところに到達するまでは、デバッギを通常速度で実行できるという利点がある。プログラマがデバッギの実行を制御するという観点では、ステップ実行は active であり、 breakpoint は passive であると言えよう。

2.2 watchpoint

breakpoint と同様、 passive な制御方法と言えるデバッガの機能に “watchpoint” がある。これはメモリ中のある領域の値が変更されたときに、デバッギの実行を停止してプログラマに知らせるというものである。breakpoint と watchpoint はいずれもデバッギの持つメモリ領域を監視しており、その領域が命令列であるかデータであるかの違いであると言える。

C のように aliasing が起こる言語では watchpoint が有効な局面は多く、典型的には「配列のオーバーラン」がこれに当たる。このような場合、破壊してしまったことによる影響は時間的にはずっと後に生じることが多く、ステップ実行や breakpoint などのソースコードのコンテクストに依存した実行制御では一般に原因の発見が困難である。これに対し、 watchpoint はメモリ領域そのものを監視するので、こうした「意図しない破壊」には有効である。

3 reverse watchpoint

それでは、実際に watchpoint を用いてデバッグする際の手順はどのようになるであろうか。

エラーを発見すると、プログラマはまずデバッギの実行を先頭からやり直し、最初のところで停止させて、監視対象のメモリ領域に watchpoint を設定する。そして実行を再開すると、対象領域への書き込みが起こる度にデバッガは停止して、値の変化と書き込んだコード（に対応するソースコードの行）を報告する。プログラマはそれを監視しながら、本当に破壊を引き起こした書き込みが起こるまで、デバッガに継続を指示し続ける。そして明らかに値がおかしい場所を発見するか、エラーを発見した場所にたどり着いたら、その時点での直前の書き込みが怪しいと判断してデバッギを再度先頭から実行し直し、問題の書き込みが起こった場所まで制御を移動させる（図 1）。

しかし、これをプログラマが手動で行うには困難な点がある。まず第 1 に、一般には何度も起きる watchpoint による停止を全て監視するのは現実的ではない。第 2 に、直前の watchpoint 停止点に注目すべきだと分かったとしても、その場所に制御を移すことは困難である。停止点に breakpoint を設定しても、その

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

(2)    init_heap();
(3)    busy_loop();
(4)    area1 = simple_alloc(4);
(5)    busy_loop();
(6)    for(i = 0; i < 3; i++){
(7)      busy_loop();
(8)      area2 = simple_alloc(4);
(9)      busy_loop();
(10)     simple_free(area2);
(11)     busy_loop();
       }
(12)   busy_loop();
(13)   area2 = simple_alloc(4);
(14)   busy_loop();
(15)   string_copy(area1, "abcdefghijk");
(16)   area3 = simple_alloc(4);
(17)   return(0);
     }
```

図 2: メモリ確保プログラム

場所は複数回実行されるために何度も停止してしまうので、プログラムの実行過程における一時点を正確に特定して、手動でその場所を特定することは大変難しい。

そこで、図 1に示した手順をデバッガによって自動化できれば、プログラマに対する有力なサポートとなり得る。これに必要なことは以下の 2 つである：

- watchpoint の履歴を取ること（Pass 1）
- Pass 1 での最後の watchpoint 停止点に制御を移動すること（Pass 2）

これらの実装については第 5節で述べることとし、次節では reverse watchpoint が有効な場合を、あるプログラムのデバッグ例を通して説明する。

4 デバッグセッション例

本節では、バグを持ったあるテストプログラムのデバッグを reverse watchpoint と通常の watchpoint で行い、reverse watchpoint の有効性を示す。

4.1 デバッギ

ここで扱うプログラムは C の `malloc` のような memory allocator を用いている。関数 `simple_alloc` は引数で指定された大きさの領域をヒープから確保して、そのアドレスを返す。`simple_free` は指定された領域をヒープに返し、再利用できるようにする。このプログラムの `main` 関数を図 2に示す。`busy_loop()` は後に行う経過時間測定で意味のある値を得るためだけに挿入してあり、ループを空転するだけの関数である。

このプログラムでは、最初に 4 バイトの領域を `area1` に確保する（4）。次にまた別の 4 バイトの領域を `area2` に確保して（8）解放する。これを 3 回繰り返した後、再び `area2` を確保する（13）。さらに

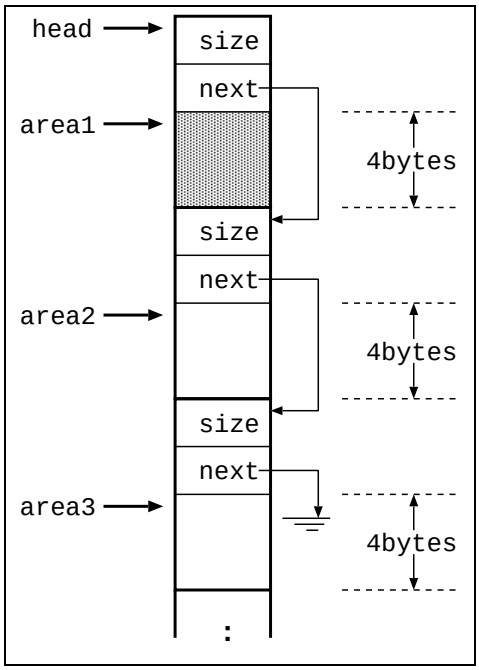

図 3: 期待されたヒープの構造

string_copy によって area1 に文字列をコピーすると（15）、文字列が与えられた領域を越えてしまうために area2 のヘッダ領域を破壊してしまう。そして、 area3 に領域を取って終了する。

area3 を確保直後のヒープの状況は図 3のようになるはずであったが、このプログラムは segmentation fault を起こしてしまう。そこで、デバッガを使ってこのバグの原因を調べてみることにする。

4.2 エラー発生点

デバッガの制御下でこのプログラムを動かしてみると、 area3 を確保すべく、 simple_alloc を図 2の（16）で呼び出したところでエラーが起こっていることが分かる。

simple_alloc は確保される領域を線形リストとして管理していて、 first-fit で新しい領域を割り当てる。そのため、 area3 は area2 の直後に取られるはずである。 simple_alloc は空き領域を探すために head から next フィールドをたどっていき、ちょうど area2 の next にアクセスしたところであった。 area2 の管理領域情報を調べてみると、以下のような結果が得られた：

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

area2 は 4 バイトであるはずなのに巨大なものになっていて、リストの最後尾のはずなのに次のセルがひどく遠くにあることになっている（図 4）。 area2 の管理領域情報が、何らかのオペレーションによって破壊されたと思われる。

4.3 reverse watchpoint によるデバッグ

問題の領域を最後に更新したオペレーションを知るために、 reverse watchpoint を使うことにする。 area2 の管理領域（0x804af9c）に対して reverse watchpoint を実行すると、以下のような結果が得られる：

```
Old value = 6776421
New value = 1751606885
```

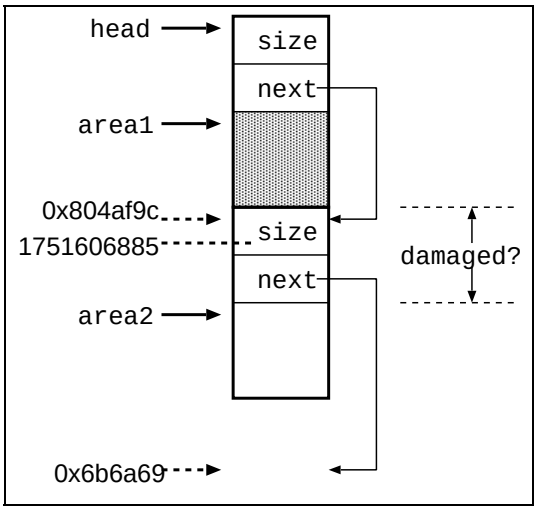

図 4: 実際の壊れたヒープ

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

これにより、最後のオペレーションは（15）で呼び出された `string_copy` の中にあり、バグの原因が 4 バイトしかない領域に書き込んだデータが長過ぎたのだと分かる。

4.4 通常の watchpoint によるデバッグ

もし reverse watchpoint が使えないと、このデバッグは面倒になる。最初に通常の watchpoint を設定してからデバッグを始めると、対象の領域はループが回る度に更新されるのでデバッギは何度も停止してしまう。プログラマはその度に、表示された 0x804af9c の値を調べて、デバッガに続行を指示しなければならない。

実際、`string_copy` による破壊は watchpoint の停止の 8 回目から 11 回目に当たっているので、最初の 7 回は無駄足を踏まされることになる。デバッギがこのようなごく簡単なテストプログラムではなく現実的な規模のプログラムであったら、プログラマの負担は計り知れない。

reverse watchpoint を使うことで、注目する領域に最後に書き込んだオペレーションを容易に捕捉できるようになる。前述の例の場合では、通常の watchpoint では大量のデバッガ・コマンドが必要となるところを、1 つのコマンドで済んでいる。

5 reverse watchpoint の実装

reverse watchpoint を実現するためには、第 3節で述べたように

- （Pass 1） watchpoint の履歴の取得
- （Pass 2）最後の watchpoint 停止点への移動

が必要となる。 Pass 1 の実現は自明であり、問題は Pass 2 の実現である。本節では、 Pass 2 の実現に必要な “dynamic breakpoint” を導入し、それから reverse watchpoint の実装について述べる。

5.1 dynamic breakpoint

5.1.1 static breakpoint と dynamic breakpoint

第 2節で述べたように、ブレークポイントはソースコードの行に対応する命令列を監視するものであり、ソースコードでの位置を指定するものである。そこでこれを *static* breakpoint （S.B.P.）と呼ぶことにする。

S.B.P. を設定した行が、ループの中や再帰関数のように繰返し呼び出される関数の中にあると、その S.B.P. に何度も当たってしまう。しかし、実行過程の中のある一時点で実行を止めたいので、これら複数のヒットを区別しなくてはならない。そのような、同じ行の他のヒットを区別できる breakpoint を、*dynamic* breakpoint（D.B.P.）と呼ぶことにする。

D.B.P. があれば、watchpoint がヒットした最後の場所に、制御を直接移動することができる。

5.1.2 dynamic breakpoint の実装

ある行が繰返し実行されるのは、それがループの中にあるか、繰返し呼び出される関数の中にあるかのどちらかである。従って D.B.P. を実装するには、関数の何度目の呼び出しか、ループの何回目の実行か、といった情報を保持する必要がある。

ところが、このような情報をデバッガが取得するには、ステップ実行のような active な実行制御を行わなければならない。breakpoint のような passive な実行制御では、デバッギはほとんどの時間を通常速度で走行しており、デバッガ側で取得・管理することはできない。そこで、「何回目の関数呼び出しか？」「何回目のループの実行か？」といった情報はデバッギ側で蓄積することにして、デバッガは breakpoint によって制御を手に入れた時に、必要に応じてそれを取得すれば良い。

ここで、デバッギに大域変数 “timestamp” を導入する。この値はプログラムが起動されたときは -1 であり、次の条件のいずれかに合致した場合にインクリメントされる：

- 関数呼び出しの入口
- 関数呼び出しの出口
- 上方向へのジャンプ
  - ループ本体の最後
  - 上向きの `goto`
  - `continue` オペレータ

図 2のコードに timestamp を更新する `INC_TS` を挿入したコードを図 5に示す。

こうすることでプログラム中のある行の実行は unique な timestamp 値と対応づけられるので、プログラム実行過程における一点は「timestamp」と「行番号」の組を用いて特定することができる。D.B.P. はまさにこの 2 つの値の組のことに他ならない。

D.B.P. は条件付 breakpoint を用いれば容易に実装できる。その条件として timestamp の値を与えれば良い。しかし条件付 breakpoint を用いると、その breakpoint にヒットする度にデバッギが停止して、デバッガがその条件の成否を判定するので、大変遅くなってしまう。というのも、デバッガ・デバッギ間のタスク切替が生じるからである。

5.1.3 D.B.P. のより効率的な実装

図 5の定義の代わりに図 6で示す `INC_TS` を用いて、D.B.P. をデバッガ制御コマンドとして実装する。`inc_ts_stop` は timestamp が目標値と一致する時のみ呼び出される。D.B.P. が指示されると、制御コマンドは以下を実行する：

1. デバッガで変数 `ref` に目標値を設定し、`inc_ts_stop` に通常の breakpoint（S.B.P.）を設定し、デバッギの実行を継続する。
2. デバッギがその S.B.P. で停止したら、目標の行に新たに S.B.P. を設定して、実行を継続する。

こうすると、デバッギを D.B.P. を設定した場所で停止させられる。実際には通常の S.B.P. しか使っていないので、デバッガとのやり取りを除いては通常速度で実行できる。

この方法であれば、性能低下のほとんどない D.B.P. を実現できる。

5.2 reverse watchpoint の実装

先に述べた D.B.P. を用いれば、reverse watchpoint の実装は単純になる。即ち、reverse watchpoint が指示された時（この場所をエラー発生点と呼ぶことにする）、以下の手順を行えば良い（図 7）：

```
     #define INC_TS (++timestamp)
     int main(void){
(1)    int i;

(**)   INC_TS;
(2)    init_heap();
           :
(5)    busy_loop();
(6)    for(i = 0; i < 3; i++){
(7)      busy_loop();
           :
(10)     simple_free(area2);
(11)     busy_loop();
(**)     INC_TS;
       }
(12)   busy_loop();
           :
(16)   area3 = simple_alloc(4);
(**)   INC_TS;
(17)   return(0);
     }
```

図 5: timestamp 更新コードを追加したメモリ確保プログラム

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

int inc_ts_stop(void){
}

#define INC_TS
  ((++timestamp == ref) && inc_ts_stop())
```

図 6: INC_TS の改良版

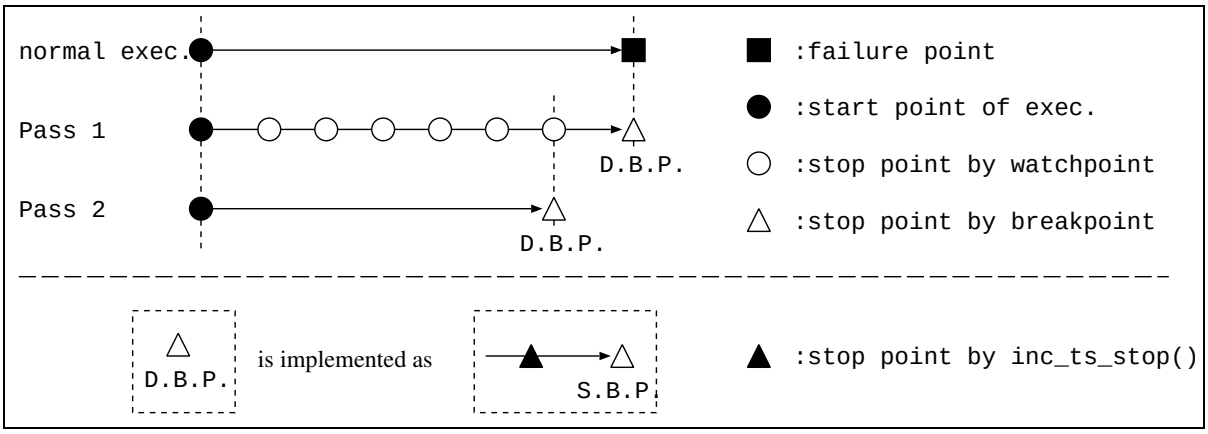

図 7: reverse watchpoint の詳細

1. デバッギを停止させ、エラー発生点での timestamp と行番号を記録し、先頭から実行し直して `main` 関数の入口で停止させる。
2. 通常の watchpoint を対象のメモリ領域に設定し、エラー発生点には D.B.P. を設定する。
3. (Pass 1) 実行を継続し、
   (a) watchpoint で停止したときは、そのときの D.B.P. データ (timestamp と行番号) を記録する。
   (b) エラー発生点の D.B.P. で停止したときは、次のステップへ進む。
4. (Pass 2) 先頭から再度実行し直し、`main` の入口で停止させ、Pass 1 で得た情報を用いて最後の watchpoint 停止点に D.B.P. を設定し、その D.B.P. まで実行を継続する。

制御がエラー発生点に到達しない限り「最後の」 watchpoint 停止点は分からないので、目標点を見つけるための Pass 1 と、そこに制御を移すための Pass 2 との、2 パスが必要になる。

5.3 必要な準備

reverse watchpoint を利用するには、以下を自動化しなくてはならない：

- `INC_TS` の挿入
- 上の手順にしたがってデバッガを制御すること

ここでは、 reverse watchpoint の有効性と実現可能性を論じるのを主眼としているので、 `INC_TS` の挿入は手動で行い、デバッガの制御はオーバーヘッド計測のための Perl スクリプトを用意した。なお、デバッガには GDB 4.18 を使用した。

6 オーバーヘッドの計測

この節では、 reverse watchpoint 実行のオーバーヘッドを計測する。

6.1 経過時間

`bash` ビルトインの `time` コマンドを用いて、以下の 3 種類の経過時間を計測した (図 8) :

1. (通常実行) デバッギを先頭から `segmentation fault` で停止するまで通常実行。
2. (direct) Perl を起動し、Perl スクリプトから GDB を起動し、GDB からデバッギを実行し、通常実行と同じことをする。
3. (rwatch) direct と同じことをし、デバッガに図 4の 0x804af9c に対する reverse watchpoint を指示する。

第 2、第 3 の場合は、デバッガとデバッギのやり取りを自動化するため、 Expect モジュールを用いた Perl スクリプトを作成した。第 2 の場合は Perl と GDB の起動にかかるコストを示すために挙げてある。第 3 の場

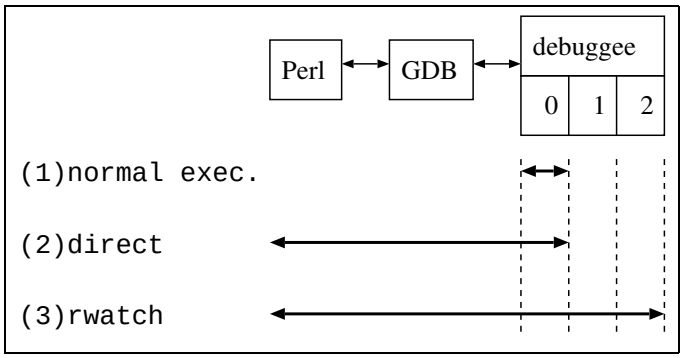

図 8: 各テストケースの詳細

表 1: reverse watchpoint の経過時間

	経過時間 (秒)
通常実行	2.756
direct	5.406
rwatch	12.882

合では、デバッギは通常実行と、 watchpoint の履歴を取る Pass 1 と、目標点へ制御を移動する Pass 2 との、 3 回の実行が行われていることに注意されたい。

これらの経過時間の測定は、 Pentium 90MHz、 40MB メモリ、 Linux 2.2.13 という環境の PC で行った。 Pentium プロセッサには watchpoint のハードウェアによるサポートがある。結果を表 1に示す。

6.2 オーバーヘッドの分析

Pass 1 と Pass 2 にかかる時間は以下の 3 つに分けられる：

1. Perl と GDB の起動
2. 2 回の通常実行
3. デバッガ・デバッギ間のやり取り

これは以下のような式でも表すことが出来る：

$$T = C + 2T_1 + NT_2 \tag{1}$$

C : 上の 1. にかかる時間
T_1 : 通常実行にかかる時間
T_2 : 上の 3. でデバッガ・デバッギ間の 1 回のやり取りにかかる時間
N : Pass 1 における watchpoint による停止回数

ここで、式 (1) の NT_2 を「オーバーヘッド」と呼ぶことにすると、第 6.1節のテストケースでのオーバーヘッドは以下のように見積もることができる：

$$12.882 - 5.406 - 2.756 \times 2 = 1.964\text{秒}$$

T_2 は定数と考えられるので、オーバーヘッドは N に比例するはずである。これを確かめるために、、図 2の (6) のループ回数を増やし、 watchpoint による停止回数を最初の 11 回から、 21、 31、 41、 51 回と増やしたときのオーバーヘッドを計測した。結果を図 9に示す。

7 逆実行に関する関連研究

再現性のないプログラムに対しては、古来から行われている履歴保存型の方式を採用せざるを得ない [2]。この方式では、言語処理系を変更ないしは作成することが必須であり、保存する履歴も大きくなりがちである。この履歴の量を少なくしようと言う試みもある [3]。

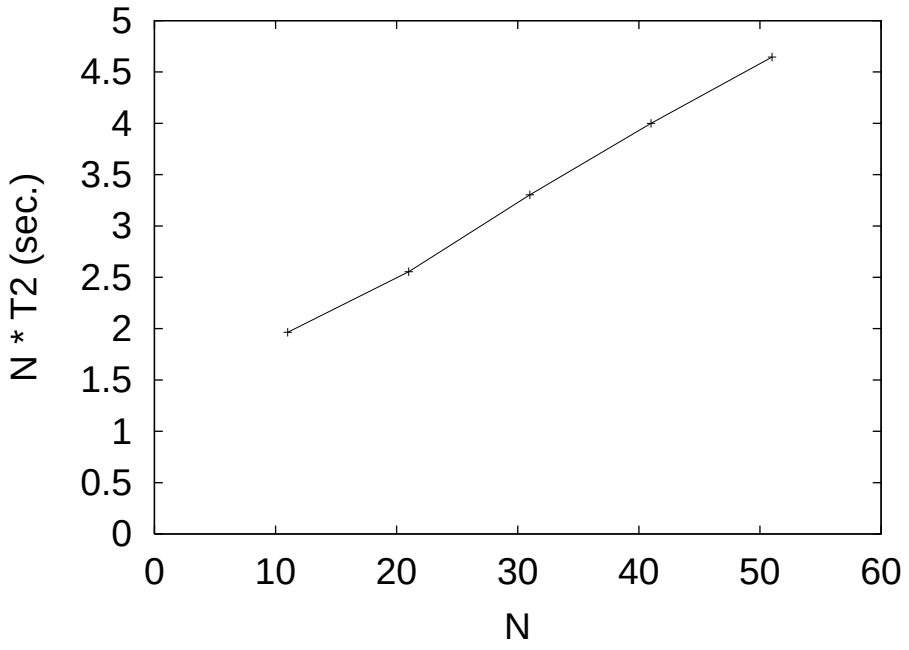

図 9: オーバーヘッドの変化

本論文では watchpoint の逆方向化について扱ったが、我々はステップ実行の逆方向化についての研究も行っている [4][5]。これらの研究では、再現性のあるプログラムを対象として、先頭から再実行することによる、関数単位の疑似的な逆実行を実現している。本論文で述べた D.B.P. を用いることにより、逆実行の単位をソーステキストの行にすることができるであろう。

8 今後の課題

- タイムスタンプ更新機構のコンパイラへの実装 — 今回はオーバーヘッド計測のためだけの目的だったので `INC_TS` の挿入を手動で行ったが、一般に利用するには GCC などのコンパイラに組み込んで挿入を自動化する必要がある。
- ユーザインタフェースの実装 — 実際のデバッグセッションで利用するには、 Emacs の `gud-mode.el` のような一般的なインタフェース（フロントエンド）にラッパーとして実装するか、 GDB にコマンドとして実装する必要がある。

9 まとめ

本輪講では、意図しないメモリ領域の破壊に対処する手法として、その場所への直前の書き込みを捕捉する reverse watchpoint を提案した。これを用いることで、同じ場所へ繰返し書き込みが発生し、従来型の watchpoint では問題の箇所の特定が困難になる場合でも有効なことを述べた。

そして、 reverse watchpoint を実現するために必要な dynamic breakpoint を導入してその実装について述べ、それを用いて reverse watchpoint を実現する方法について述べた。

また、 watchpoint の履歴取得によるオーバーヘッドが watchpoint の停止回数に比例することを示すことで、現実的な時間で動作することを示した。

A プロセッサによるサポート

今回利用している watchpoint の仕組みは、ハードウェア（プロセッサ）のサポートがない場合、 breakpoint に比べてひどくオーバーヘッドが大きいものになってしまう。というのも、デバッギがメモリに書き込

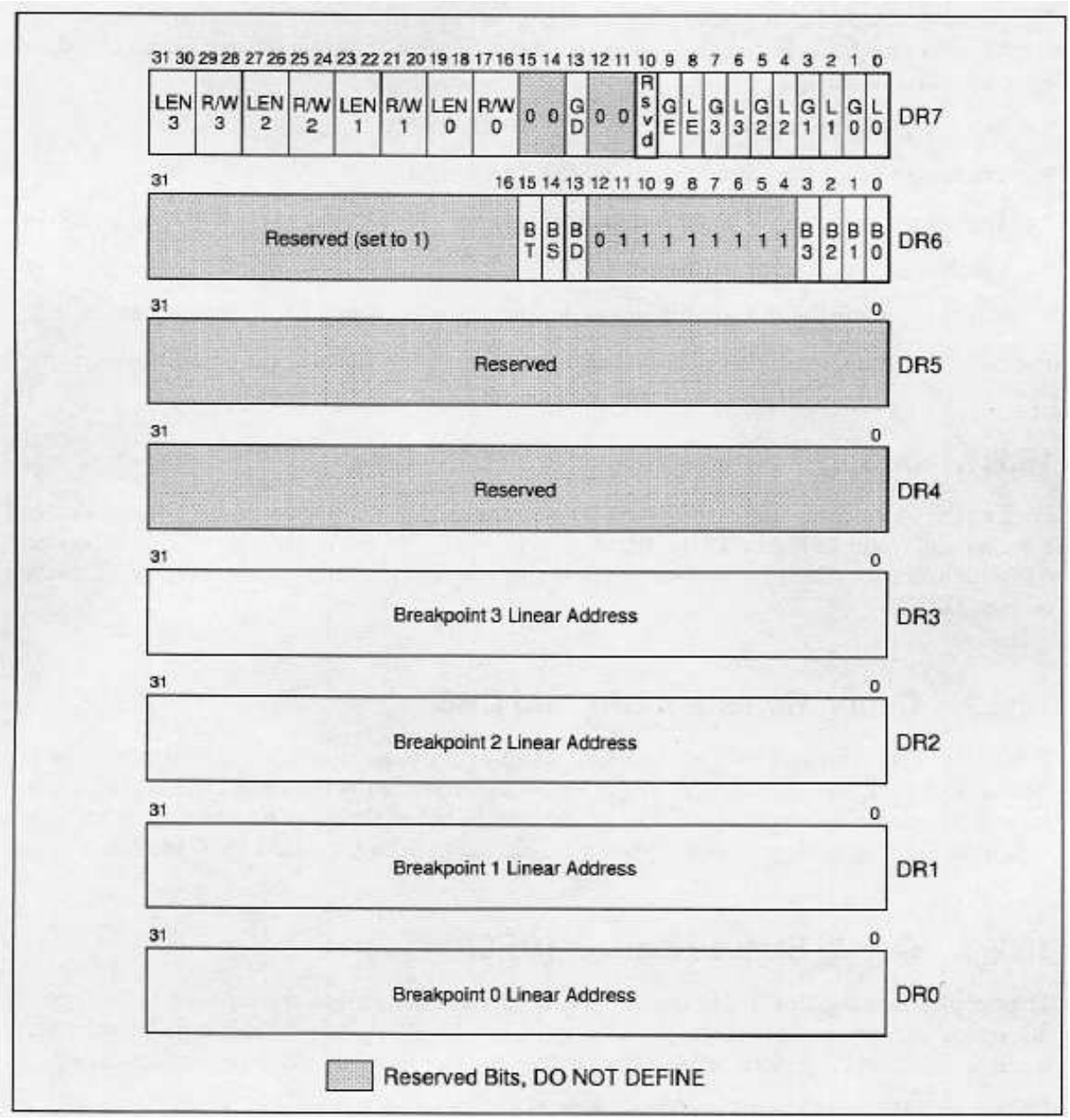

図 10: IA-32 のデバッグレジスタ

むたびに、監視している領域の値が変化していないか調べなければならないからである。こうしたサポートを持ったプロセッサとして、Intel Architecture について紹介する。

現在流通している IA-32[6] に則したプロセッサは、breakpoint として使えるレジスタを 4 つ（DR0 ～ DR3）持っている（図 10）。それぞれのレジスタにはメモリ中のアドレスを指定することができ、コントロール・レジスタである DR7 の R/W で捕捉するオペレーションの対象（命令・データ・I/O）と種類（読み・書き）を、LEN で監視する領域の大きさ（1・2・4 バイト）を指定できる。デバッグ・レジスタに指定したオペレーションが行われると、プロセッサは Debug Exception（#DB）を発生させて通知する。debug address register が 4 つあるので、ハードウェアサポートのある breakpoint と watchpoint は、合わせて 4 つまで利用可能ということになる。

今後登場する 64-bit architecture である IA-64[7] では、breakpoint のための IBR（Instruction Breakpoint Register）と watchpoint のための DBR（Data Breakpoint Register）が、アドレス指定用と条件指定用の組で 4 組ずつ提供されることになっている（図 11、図 12）。

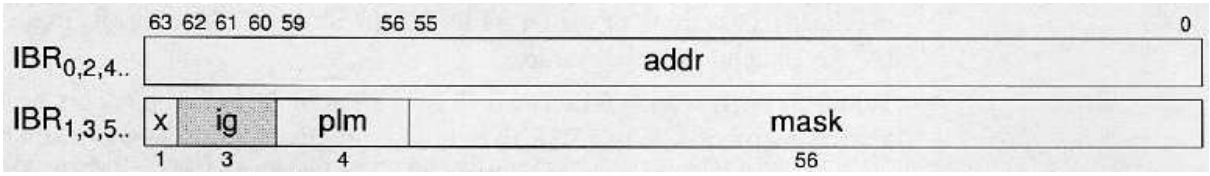

図 11: IA-64 の Instruction Breakpoint Register

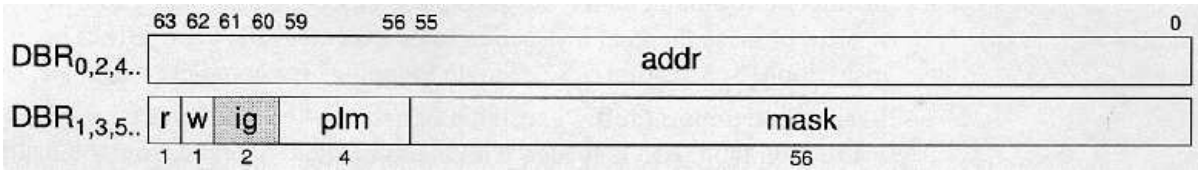

図 12: IA-64 の Data Breakpoint Register

参考文献

[1] Jonathan B. Rosenberg. *How Debuggers Work.* John Wiley & Sons, Inc., 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] 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.

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

[4] 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, 2000. (掲載予定).

[5] 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, 2000. (投稿中).

[6] *Intel Architecture Software Developer's Manual Volume 3: System Programming*, 1999.

[7] *Intel IA-64 Architecture Software Developer's Manual Volume 2: IA-64 System Architecture*, 2000.