平成12年度冬学期 第1回情報工学輪講

Reverse Watchpoint

寺田研究室 M2 丸山 一貴
kazutaka@sanpo.t.u-tokyo.ac.jp

Contents

1. はじめに

2. reverse watchpoint

3. デバッグセッション例

4. dynamic breakpoint

5. reverse watchpoint の実装

6. オーバーヘッド計測

7. 関連研究とまとめ

はじめに

- プログラムのバグ ： 2種類

- static に発覚するバグ
  - syntactic なもの、symbol の解決など
  - 原因の特定・修正は比較的容易

- dynamic に発覚するバグ
  - 実行時に発覚
  - デバッグ対象プログラム (デバッギ) の実行状態が不明
    - 変数の値
    - エラーの起きた場所
  - 一般に、原因の特定が困難

↓

デバッガの利用

aliasing

- デバッガでも分かりにくいバグ

- 変数の値がおかしい
  - その変数への代入文
  - その変数領域を指すポインタ
    - 配列のオーバーラン

- aliasing
  - 変数に別名 (==ポインタ) がある
  - その変数への代入文だけでは分からない

↓

watchpoint （ data access breakpoint ） の利用

watchpoint

- あるメモリ領域の書き換えを捕捉
  - breakpoint 同様、passive な実行制御
- breakpoint
  - ソースコードの行に対応する命令列を監視
  - 制御の到達を捕捉
- watchpoint は breakpoint のデータ領域対象版
- 効率良い実装にはハードウェアサポートが重要
  - ソフトウェアでやると非常に遅い
  - Pentium (Intel Architecture) にはある

Contents

1. はじめに

2. reverse watchpoint

3. デバッグセッション例

4. dynamic breakpoint

5. reverse watchpoint の実装

6. オーバーヘッド計測

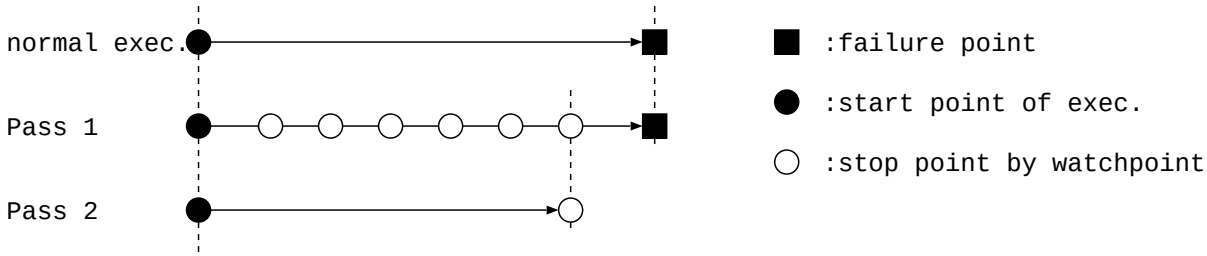
7. 関連研究とまとめ

reverse watchpoint（１）

□変数の値がおかしいとき
- ○その原因を知りたい
- ○最後に代入したオペレーションが怪しい

□手続き型・逐次型・再現性のあるプログラムを対象
- ○先頭からやり直せば、同じ状況を再現できる

□watchpoint を用いたデバッグ



reverse watchpoint（2）

- 手動で行うのは困難
  - watchpoint は何度も停止する
  - 最後の停止点を特定できない
    - 同じ行は何度も実行される
    - 実行過程の1点を特定する必要

- 処理の自動化
  - Pass 1：watchpoint の履歴取得
    - 実現は自明
  - Pass 2：最後の watchpoint へ移動
    - dynamic breakpoint の導入（後述）

Contents

1. はじめに

2. reverse watchpoint

3. デバッグセッション例

4. dynamic breakpoint

5. reverse watchpoint の実装

6. オーバーヘッド計測

7. 関連研究とまとめ

デバッグセッション例

- デバッギ：メモリ確保プログラム
  - heap
    - メモリのプール
  - simple_alloc
    - heap からメモリ取得
    - 線形リストで管理
    - first-fit で取得
  - simple_free
    - heap に返す

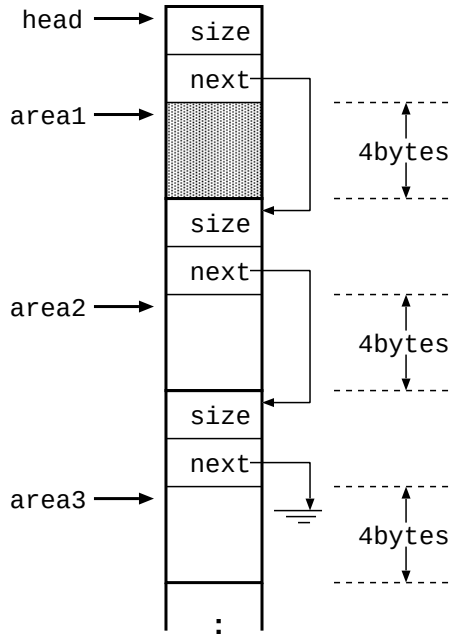
- 実行すると segmentation fault

デバッギ

```
    int main(void){
(1)    int i;
(2)    init_heap();
(4)    area1 = simple_alloc(4);
(6)    for(i = 0; i < 3; i++){
(8)      area2 = simple_alloc(4);
(10)     simple_free(area2);
       }
(13)   area2 = simple_alloc(4);
(15)   string_copy(area1, "abcdefghijk");
(16)   area3 = simple_alloc(4);
```

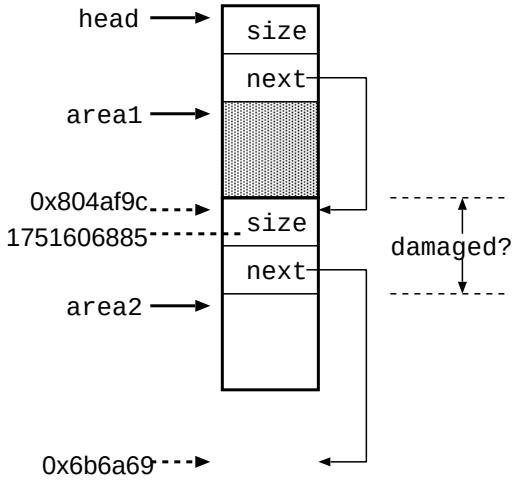
期待される動作

- area2 の直後に area3 が確保される



エラー発生点

- デバッガにかけると
  - (16) の simple_alloc でエラー
  - area3 のための空き領域探索中に発生
  - area2 の管理領域が破損



reverse watchpoint によるデバッグ

- reverse watchpoint を使うと

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

- string_copy で書いた文字列が長過ぎた

通常の **watchpoint** によるデバッグ

- area2 に書き込まれるたびに停止
  - 全部で11回停止する
  - string_copy による破壊は8回目から11回目
  - 少なくとも7回は無駄足を踏む

↓

現実的な規模のプログラムでは
通常の watchpoint で
先頭から追跡するのは困難

↓

reverse watchpoint の方が有効

Contents

1. はじめに

2. reverse watchpoint

3. デバッグセッション例

4. dynamic breakpoint

5. reverse watchpoint の実装

6. オーバーヘッド計測

7. 関連研究とまとめ

dynamic breakpoint

- 通常の breakpoint
  - ソースコード上の1点
  - これを static breakpoint (S.B.P.) と呼ぶことにする
  - 一般には複数回ヒット
    - ループ内部、繰返し呼び出される関数の内部

- 実行過程における1点を特定したい
  - 上記の「複数回ヒット」を区別
  - これを dynamic breakpoint (D.B.P.) と呼ぶことにする

↓

そのループ本体の何度目の実行か？
その関数の何度目の呼び出しか？
これらが分かれば良い

デバッガで知るには...

- ループ実行回数・関数呼び出し回数
  - ステップ実行 (active な実行制御) なら分かる
    - 処理が遅過ぎる
  - breakpoint (passive な実行制御) では分からない
- デバッギ自身に数えさせる
  - 数えるための変数を用意
  - デバッギは自身で変数を更新
- デバッガは必要に応じてその値を読む

timestamp の導入

- 大域変数 timestamp を導入
- 初期値は -1、以下の条件でインクリメント
  - 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

↓

プログラムのある行の実行には
必ず unique な timestamp 値が対応

↓

timestamp 値と行番号の組で
実行過程の1点を特定できる

デバッギ（改）

```
    #define INC_TS (++timestamp)
    int main(void){
(1)    int i;
(**)   INC_TS;                    /* 関数入口 */
(2)    init_heap();
          :
(6)    for(i = 0; i < 3; i++){
          :
(10)     simple_free(area2);
(**)     INC_TS;                   /* ループ本体末尾 */
       }
          :
(16)   area3 = simple_alloc(4);
(**)   INC_TS;                    /* 関数出口 */
(17)   return(0);
```

dynamic breakpoint の実装

- デバッガの条件付 breakpoint を使う
  - 条件：timestamp 値と目標値との比較

- しかし、条件付 breakpoint は遅い
  - デバッガが条件の成否をテスト
  - デバッガ・デバッギ間でのタスク切替

↓

速度低下のほとんどない

条件なし S.B.P. だけで実現したい

D.B.P. の効率的な実装

□INC_TS の実装を以下のように変更

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

□D.B.P. の目標値 ： timestamp と行番号

□ステップ1

- ○デバッガで ref に timestamp の目標値を設定
- ○inc_ts_stop には S.B.P. を設定

□ステップ2

- ○デバッギが inc_ts_stop で停止したら
- ○目標の行番号に S.B.P. を設定

Contents

1. はじめに

2. reverse watchpoint

3. デバッグセッション例

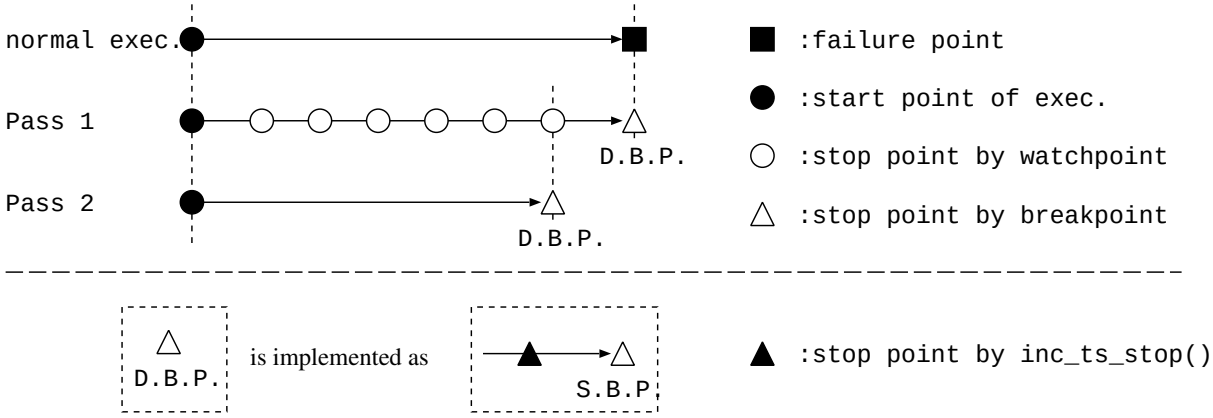
4. dynamic breakpoint

5. reverse watchpoint の実装

6. オーバーヘッド計測

7. 関連研究とまとめ

reverse watchpoint の実装



- Pass 1
  - 監視対象領域への全ての書き込みを watchpoint で取得
- Pass 2
  - Pass 1 の情報より、最後の書き込み場所に D.B.P. で移動

必要な準備

- reverse watchpoint を利用するには：
  - INC_TS の自動挿入
  - reverse watchpoint の実現手順を自動化

- 今は実現可能性とオーバーヘッドに注目
  - INC_TS は手動挿入
  - デバッガは Perl スクリプトで制御
  - 自動化は今後の課題

Contents

1. はじめに

2. reverse watchpoint

3. デバッグセッション例

4. dynamic breakpoint

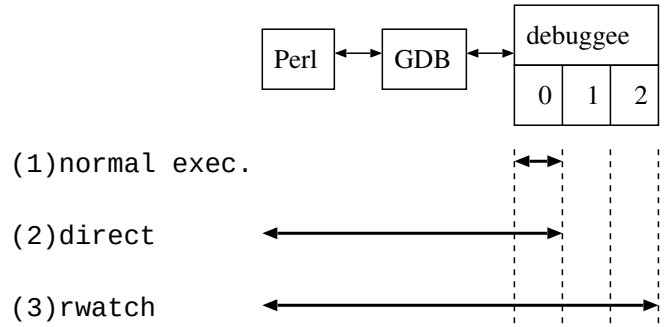
5. reverse watchpoint の実装

6. オーバーヘッド計測

7. 関連研究とまとめ

オーバーヘッド計測

- bash の time コマンドで経過時間を計測
  - 1. : 通常実行
  - 2. : 1. + Perl と GDB の起動
  - 3. : 2. + reverse watchpoint

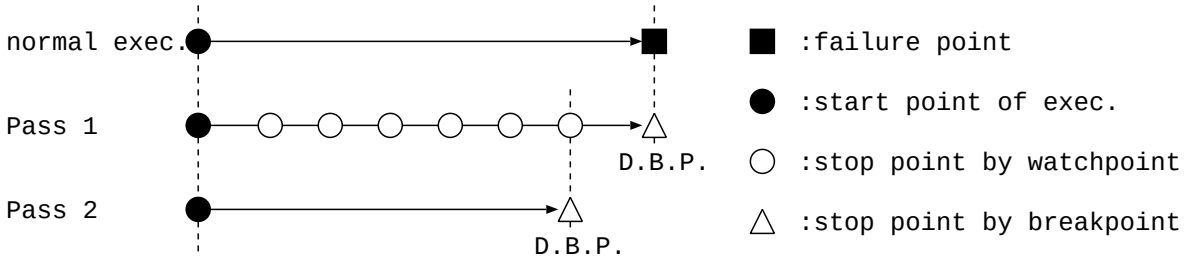


- Pentium 90MHz、40MB メモリ、Linux 2.2.13
  - Pentium は watchpoint のハードウェアサポートあり

計測結果と分析

- 1. 通常実行： 2.756 秒
- 2. direct： 5.406 秒
- 3. rwatch： 12.882秒

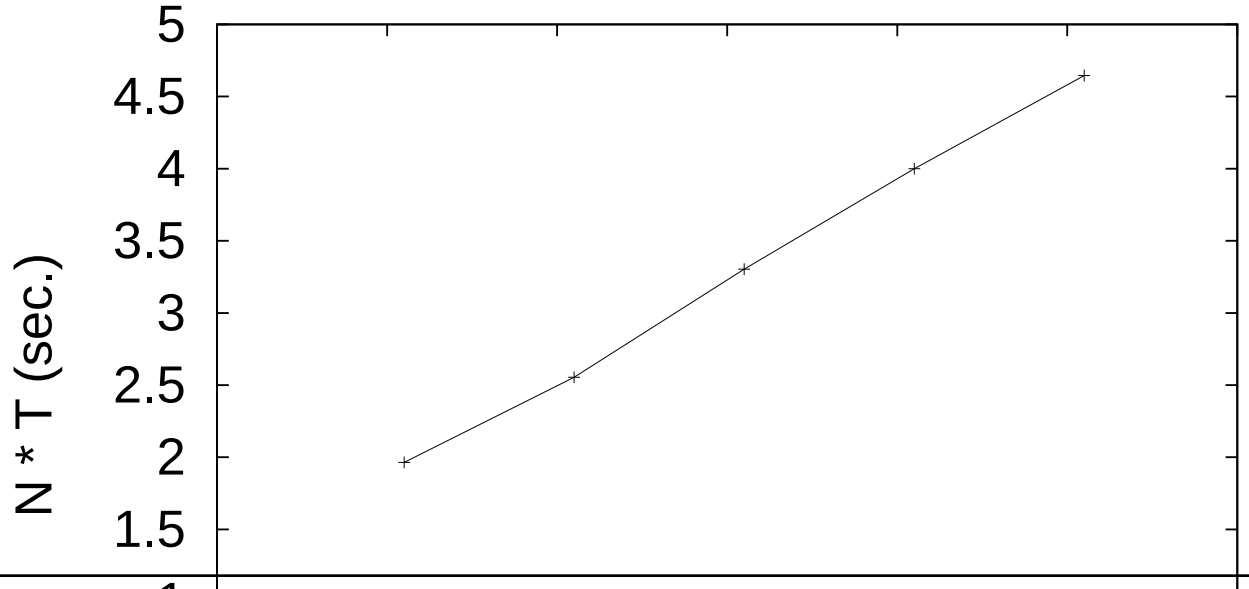
- Pass 1 と Pass 2 の時間は3つに分かれる：
  - Perl と GDB の起動
  - 2回の通常実行
  - Pass 1 におけるデバッガ・デバッギ間のやり取り
    - これをオーバーヘッドと呼ぶ



オーバーヘッドの変化

- オーバーヘッドは
  - 1回のやり取りにかかる時間：T (ほぼ定数)
  - Pass 1 における watchpoint による停止回数：N
  - N と T の積 → N に比例するはず

- watchpoint の停止回数を変える
  - デバッギのループの回数を変える
  - 停止回数：11、21、31、41、51回



Contents

1. はじめに

2. reverse watchpoint

3. デバッグセッション例

4. dynamic breakpoint

5. reverse watchpoint の実装

6. オーバーヘッド計測

7. 関連研究とまとめ

逆実行の関連研究

- 再現性のないプログラム
  - 古来よりの履歴保存型 (文献[2])
  - 履歴を減らす工夫 (文献[3])

- active な実行制御の逆実行
  - 関数単位の疑似逆実行 (文献[4][5])

まとめ

- reverse watchpoint の提案
  - ある変数をおかしくした原因を見つける
  - 従来の watchpoint よりも有効な機能
- reverse watchpoint の実装
  - 実行過程の1点を特定する
    - dynamic breakpoint の提案
    - 効率的な実装
- オーバーヘッド計測
  - watchpoint の停止回数に比例
  - 現実的な時間で動作する

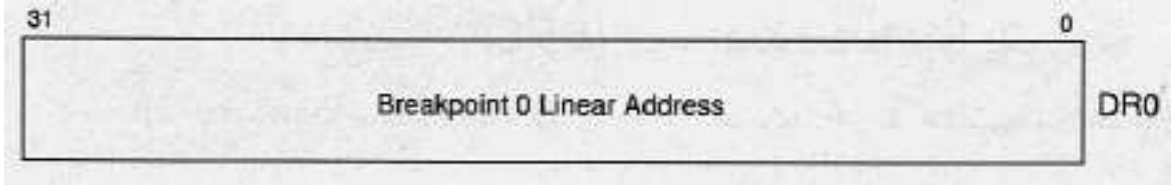
付録

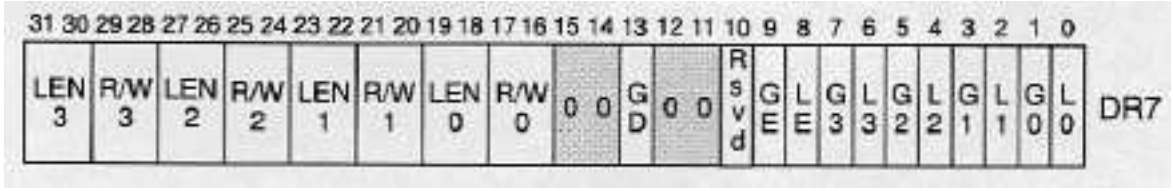
watchpoint のハードウェア・サポート

付録：IA-32 でのサポート

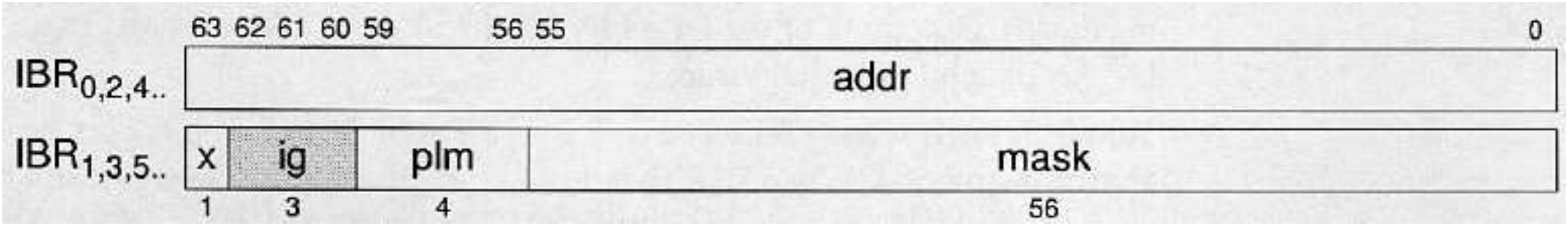
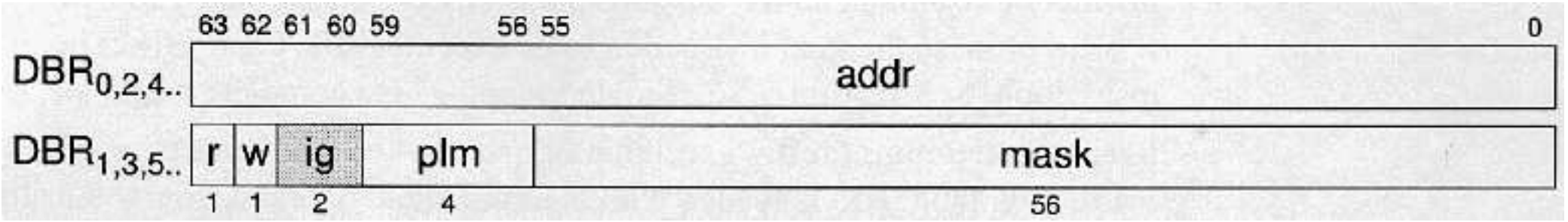
- Debug Address Registers (DR0 ~ DR3)
  - breakpoint では命令のアドレス
  - watchpoint では変数 (の領域) のアドレス

- Debug Control Register (DR7)
  - R/W：アクセスの対象と種類
    - 命令実行・I/O・書き込み・読み書き
  - LEN：対象領域の大きさ
    - 1・2・4バイト





付録：IA-64 でのサポート

- Instruction Breakpoint Register (4組)
  - breakpoint 用のレジスタ群
  - 偶数番：アドレス保持
  - 奇数番：条件保持

- Data Breakpoint Register (4組)
  - watchpoint 用のレジスタ群