ソフトウェア工学 67－4
（１９８９．７．２５）

コンパイラを用いたパッチデータ作成方法

遠城秀和　谷口秀夫　伊藤健一
NTT データ通信 (株)

ソフトウェアの生産性とともに保守性の向上を目的に、高級言語によるソースコード差分情報を利用してロードモジュールへのパッチデータを作成する方法について述べる。コンパイラが生成するオブジェクトファイル中では、一時変数や外部変数のアドレスおよび実行コードやデータのアドレスが未確定である。そのため、アドレス確定の方法として、(1) 一時変数の割り付けを同じにするスケルトンの活用、(2) ロードモジュールからの外部変数アドレス情報の抽出、(3) リンカによる実行コードやデータを置くアドレスの確定化、を提案する。さらに、既存のコンパイラを使ったパッチデータ作成の例について報告する。これによりソースコードレベルでの修正情報からコンパイラを使ってパッチデータを自動的に作成する手順を構築できる見通しを得た。

HOW TO MAKE PATCH DATA COMPILED A SOURCE CODE

Hidekazu ENJO　Hideo TANIGUCHI　Ken-ichi ITOH
NTT DATA COMMUNICATIONS SYSTEMS CORPORATION
NTT DATA Yokohama Nishi Bldg., 2-11-6 Kita Saiwai,
Nishi-ku, Yokohama, 220 JAPAN

It is presented how to make patch data compiled a source code for productivity and maintenance. It is a problem that there are three kinds of undefined addresses in a compiled object — the allocated addresses of temporal and external variables and the start address of code and data. We propose to use the skeleton code in order that temporal variables are allocated as same address as original ones, to use the address attribute of external variables defined in the load module and to use a linker for defining the start address. And the method of making patch data is described. Finally, it is possible to make patch data compiled a differential sources code automatically.

1 はじめに

現在、ソフトウェアの生産性や保守性の向上を目的としてソフトウェア開発にC言語等の高級言語が活用されている。開発時、高級言語で作成したプログラムの修正はソースコードレベルで行なわれ、効率良く進められている。しかし、利用者に提供したプログラムを修正する場合、プログラムの走行環境と開発環境が異なるという問題がある。

ソースコードレベルの修正に比べ差分情報でロードモジュールを直接書換える方法(通常パッチと呼ばれる)は、ロードモジュールをデータとして直接書換えるので走行環境に置かれたプログラムの修正に適している。しかし、従来のパッチデータの作成方法は高級言語で書かれたソースコードを対象外としているためコンパイルされたロードモジュールを機械語レベルでデバッグする必要が生じ保守効率が低下する。

本論文では、ソフトウェアの生産性とともに保守性の向上を目的として高級言語を用いてソースコードレベルで修正情報を記述し、コンパイラを用いてロードモジュールへのパッチデータを作成する方法について述べる。さらに、既存のコンパイラを使ったパッチデータ作成の実験結果について報告する。

2 走行環境での修正方法

開発環境と同一手順のソースコードレベル修正を走行環境でするには以下の条件が必要である。

1. ソースコードが全て用意されている。
2. 開発環境で用いた同一の言語系(コンパイラ、アセンブラ、リンカ)が使用できる。
3. 開発環境と同一の環境設定がされている。

利用者に提供するプログラム走行環境では上記条件を満たすことはほとんど不可能であり、開発環境と同一手順のソースコードレベル修正はほぼ不可能である。これに対しパッチによる修正方法はロードモジュールを直接書換えるため、パッチを行うプログラムが走行する環境であればどこでも行える。パッチは走行環境におけるプログラム修正方法に適するといえる。

しかし、従来のパッチデータの作成方法は高級言語で書かれたソースコードを対象外としており、パッチデータ作成にはコンパイルされたロードモジュールの機械語レベル解析が必要であった。このため、ソフトウェア開発に高級言語を用いる利点を失うため、高級言語で開発したソフトウェアの修正方法としてパッチは利用されなかった。パッチを走行環境の修正方法として有効に利用するには、高級言語で記述されたソースコードを対象とするパッチデータ作成手法の確立が必要である。

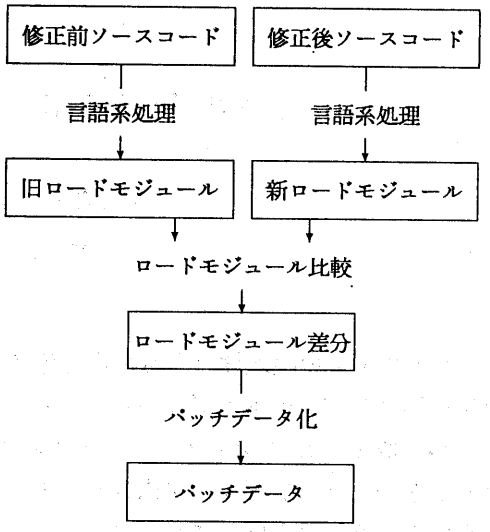

図1: ロードモジュール差分からのパッチデータ作成

3 高級言語を対象とするパッチデータ作成方法

高級言語で記述されたソースコードを機械語に変換するには、コンパイラを活用する必要がある。コンパイラを用いてパッチデータを作成する方法として、以下の2通りが考えられる。

1. 修正前と修正後のロードモジュール差分を作成し、その差分情報からパッチデータを作成する方法(図1)
2. 修正前と修正後のソースコードからソースコード差分を抽出し、その差分情報からコンパイラを使ってパッチデータを作成する方法(図2)

ロードモジュール差分を使う方法は、単純な新旧のロードモジュール差分でよく容易にパッチデータを作成できる。しかし、実行文の追加や削除を行うとロードモジュールの途中からズレが生じそれ以降全てが異ってしまう。そのため大量な差分が生成されパッチデータも大量となる。大量のパッチデータはパッチエリアの効率の面からみても実用上問題となる。

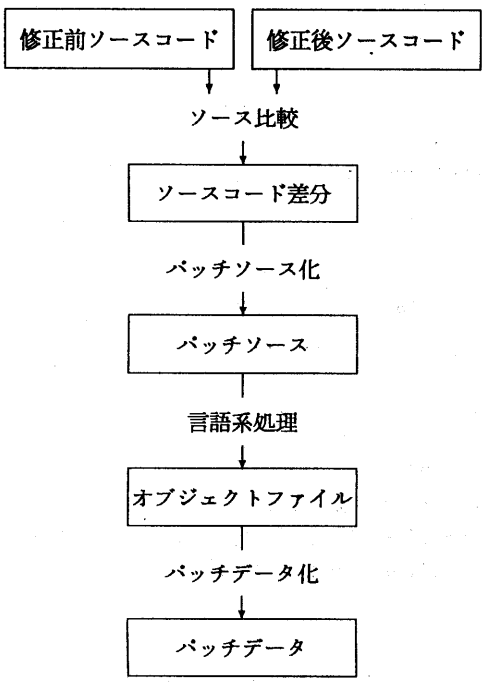

図 2: ソースコード差分からのパッチデータ作成

ソースコード差分を使う方法では差分情報だけのパッチデータが作成でき、不要なパッチデータが生成されない利点がある。そこで本方法ではソースコード差分を作成し、それを元にパッチデータを作成することとした。

4 C 言語上の実現例

UNIX[1]に代表される C 言語のコンパイラは、まず C 言語で書かれたソースコードからアセンブラソースに変換する。その後コンパイラはアセンブラを起動してアセンブラソースからオブジェクトモジュールを作成し、さらにリンカを起動してロードモジュールを作成している。

コンパイラが生成するオブジェクトファイルを逆アセンブルしパッチデータを作成する場合、アドレスに関し以下の問題を解決する必要がある。

1. スタック上に取られる一時変数のアドレス確定
2. 外部参照変数のアドレス確定
3. 追加する実行コード部分およびデータ部分の先頭アドレス確定

[1]UNIX は AT&T のベル研究所が開発した OS です。

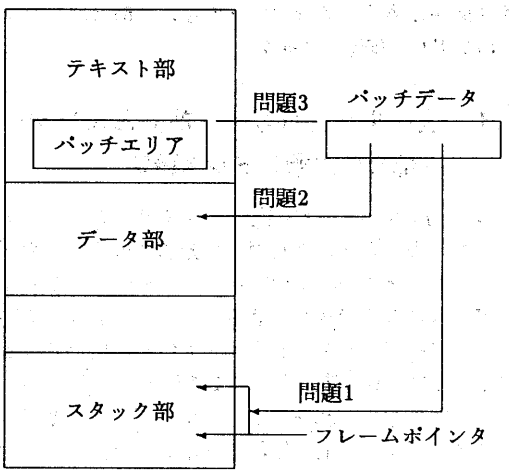

図 3: パッチデータにおけるアドレス確定

4.1 スタック上に取られる一時変数のアドレス確定

スタック上に取られる一時変数のアドレスはコンパイル時に決定され、フレームポインタからの相対アドレスとして参照される。同様に、関数引き数もフレームポインタからの相対アドレスで参照される。割り付けは一時変数や引き数とも記述した順序に従う。

そこで、修正をする部分を含む関数と一時変数や引き数の記述順序および名前を同一にすれば、追加実行文中で用いる名前の割り付けを修正される関数と同じできる。具体的には、一時変数や引き数の記述部分だけを抽出したソースコード (スケルトン) を作っておき、後から差分情報を追加することとした。

4.2 外部参照変数のアドレス確定

C 言語コンパイラは外部参照をシンボルとして扱い、属性情報をシンボルに付加するアセンブルソースを生成するだけである。アセンブラも属性情報をリンカの使用形式に変換するのみでリンカが実際のアドレスと対応づける。このため、外部参照シンボルはオブジェクトファイル中でもアドレス値が未定義である。一方、当然のことながらロードモジュール中では外部参照とアドレスは対応づけられ、外部参照シンボルとアドレスの対応情報は抽出可能である。

そこで、ロードモジュールから外部参照シンボルとアドレスの対応情報を抽出し、既確定のアドレス

属性情報の形式でアセンブラに与え外部参照シンボルのアドレス確定をさせた。

4.3 追加する実行コード部分およびデータ部分の先頭アドレス確定

C言語コンパイラは実行コードやデータを置くアドレスの決定をリンカ処理まで先送りにする。オブジェクトファイル中では実行コードやデータを置くアドレスは未確定状態である。しかし、追加する実行コードやデータがパッチエリアの未使用部分に置けるようアドレス確定化が必要である。

そこで、与えられたアドレスに実行コードやデータを置くリンカ一般の機能を利用し、実行コードやデータを置くアドレスをリンカへに指示してオブジェクトファイルを変換した[1]。

5 パッチデータ作成実験

図4に示すサンプルプログラムに対し 2行目のprint文を図5に示すように修正するパッチデータ作成実験を行った。

パッチデータは以下の2部分から構成される。

1. 新たなソースコード追加に相当するデータ
2. 不要になった部分から追加するパッチデータに制御を移すジャンプ命令に相当するデータ

ここでは、新たなソースコードを追加するパッチデータの作成方法について述べる。

5.1 パッチデータ作成実験手順

1. パッチエリアの事前確保

   パッチエリアを確保するダミーソース (図6) のオブジェクトをパッチするロードモジュールに事前にリンクしておく。

2. 前準備

   (a) スケルトンの作成

   関数 func は char へのポインタ型の引数 s を持つ1引数関数で int 型の外部変数 a と int 型の一時変数 b を参照する。そこで、図7に示す同様な a、s、b の定義を持つ関数 dummy をスケルトンに用いる。

```
int     a;

main()
{
  func("abc");
}

func(s)
char    *s;
{
int     b;

  a = 10;
  b = 100;
  printf("%s %d %d\n", s, a, b);
  printf("---\n");   /* この部分にパッチ */
  printf("%s %d %d\n", s, a, b);
}
```

図 4: パッチを当てる元のサンプルプログラム

```
int     a;

main()
{
  func("abc");
}

func(s)
char    *s;
{
int     b;

  a = 10;
  b = 100;
  printf("%s %d %d\n", s, a, b);
  printf("===\n");      /*            */
  a = 20;               /*            */
  b = 400;              /* パッチ内容 */
  s[0] = 'z';           /*            */
  printf("===\n");      /*            */
  printf("%s %d %d\n", s, a, b);
}
```

図 5: 修正したサンプルプログラム

```
char    dum_data[256] = {0};
char    dum_bss[256];
dum_func()
{
  asm("nop;nop;nop;nop;nop;nop;nop;nop");
  asm("nop;nop;nop;nop;nop;nop;nop;nop");
  asm("nop;nop;nop;nop;nop;nop;nop;nop");
  asm("nop;nop;nop;nop;nop;nop;nop;nop");
}
```

図 6: パッチエリア用ダミーソース例

```
int     a;

dummy(s)
char    *s;
{
int     b;
}
```

図 7: スケルトン

(b) 外部参照アドレスの抽出

外部参照アドレス表示をする nm コマンド[2] の出力をアセンブラの入力情報となる asm 文の形式に編集する (図8)。

3. ソース比較

diff コマンド[2] を用いて差分情報を抽出する。(図9)

4. パッチソース化

スケルトンと diff コマンド出力を組み合わせて編集し、コンパイラで翻訳できる形式にするとともに、戻るためのジャンプ命令を最後に入れる。戻り先のアドレスは、ロードモジュールの逆アセンブラ出力から得られる無効化部分の次の実行文のアドレスとする。(図10)

5. 言語系処理

コンパイラ、アセンブラ、リンカを用いてオブジェクトファイルを作成する。この際、リンカにパッチエリアの格納アドレスを指示する。

6. パッチデータ化

逆アセンブラ出力 (図11) を編集し、パッチコマンドへの入力データの形式にする。(図12)

```
asm("set main          ,      68");
asm("set func          ,      88");
asm("set _start        ,       0");
asm("set exit          ,   10088");
asm("set a             ,   66988");
asm("set printf        ,     484");
asm("set dum_data      ,   65576");
asm("set dum_bss       ,   66992");
asm("set dum_func      ,     192");
```

図 8: コンパイラが使用可能な外部参照情報の一部

```
16c16,20
<   printf("---\n");
---
>   printf("===\n");
>   a = 20;
>   b = 400;
>   s[0] = 'z';
>   printf("===\n");
```

図 9: ソースコードの差分情報 (diff コマンドの出力)

```
#include "ExtDef.h" /* 外部参照情報 */
int a;

dummy(s)
char *s;
{
int b;

  printf("===\n");
  a = 20;
  b = 400;
  s[0] = 'z';
  printf("===\n");
  asm("jmp func+0x44");
}
```

図 10: パッチソース

```
dummy()
[1] $  c0:  480e ffff fff8          link.l  %fp,&-8
[4] $  c6:  2ebc 0001 0028          mov.l   &65576,(%sp)
    $  cc:  4eb9 0000 01e4          jsr     0x1e4.l
[5] $  d2:  23fc 0000 0014 0001 05ac  mov.l  &20,0x105ac.l
[6] $  dc:  2d7c 0000 0190 fffc     mov.l   &400,-0x4(%fp)
[7] $  e4:  1dbc 007a 0161 0008     mov.b   &122,([0x8.w,%fp])
[8] $  ec:  2ebc 0001 002d          mov.l   &65581,(%sp)
    $  f2:  4eb9 0000 01e4          jsr     0x1e4.l
    $  f8:  4ef9 0000 009c          jmp     0x9c.l
```

図 11: 逆アセンブラ出力

```
/c0
480e ffff fff8 2ebc 0001 0028 4eb9 0000
01e4 23fc 0000 0014 0001 05ac 2d7c 0000
0190 fffc 1dbc 007a 0161 0008 2ebc 0001
002d 4eb9 0000 01e4 4ef9 0000 009c/
```

図 12: パッチ用入力データ例

```
abc 10 100
---
abc 10 100
```

図 13: 修正前のサンプルプログラムの実行結果

5.2 パッチデータ作成実験結果

パッチデータ作成実験の修正前実行結果を図13に、修正後実行結果を図14に示す。

```
abc 10 100
===
===
zbc 20 400
```

図 14: 修正後のサンプルプログラムの実行結果

6 オプティマイザへの対応

高級言語、特にC言語のコンパイラではオプティマイズの機能を持つものが一般的である。オプティマイズすると高級言語で記述された実行文と生成される機械語群が必ずしも1対1に対応せず、パッチデータの作成を困難にする。しかし、修正単位を本方法で用いた実行文単位から、オプティマイズ処理によってまとめられた複数の実行文単位に変更することで同様に対応可能と考えられる。

7 まとめ

ソフトウェアの生産性とともに保守性の向上を目的にソースコード差分を基にコンパイラ利用によるパッチデータ作成方法について述べた。

コンパイラが生成するオブジェクトファイル中では、一時変数や外部変数のアドレスおよび実行コードやデータを置くアドレスが未確定である。アドレス確定対処として、一時変数を同じ割り付けにするスケルトンの活用、ロードモジュールから外部変数アドレス情報の抽出、リンカによる実行コードやデータのアドレス確定を自動化した。

その結果、ソースコードレベルでの修正情報からコンパイラを使ってパッチデータを自動的に作成する手順を構築できる見通しを得た。

参考文献

[1] SYSTEM V/68 Common Link Editor Reference Manual, Motorola Inc., 1984.

[2] SYSTEM V/68 User's Manual, Motorola Inc., 1983.