差分処理指向の言語処理系における効率的で便宜な部分実行

井上　武

法政大学工学部情報電気電子工学科

1. 前書き

本研究では、目的プログラムの変更部、及び、その関連部分のみに限定して部分実行する手法を提案する。ここでは、差分コンパイラと部分実行のアルゴリズムを融合することにより、目的プログラムの再実行時に差分コンパイルの結果をフィードバックさせる。それにより、差分コンパイル後の目的プログラム実行作業において、全ての目的プログラムを実行する必要がなくなる。そして、ユーザが差分処理を意識することなく効率的で便宜な処理が可能となる。

プログラム実行の差分処理手法として、修正した範囲のみの実行、実行結果の更新が可能な、部分実行のアルゴリズムが存在する。部分実行の例を図1に示す。

$a=1 \quad b=a+1 \quad c=1$

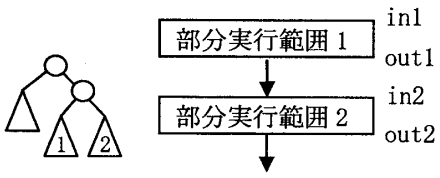
図1　部分実行が適用される例

図1において、変数aの値が修正された場合、その更新結果を用いる$b=a+1$の演算も続けて再計算すべきである。しかし、$c=1$の演算については、修正の影響を受けないので再計算が不要となる。このように、ある命令に対する修正が行われた場合、更新結果を用いる影響範囲のみを連鎖的に再計算する。

2. 目的プログラムの効率的な部分実行

差分コンパイラの差分処理手法として、解析木の各部分木ごとに、保存、再利用を行うアルゴリズムが存在する。この差分コンパイラのアルゴリズムと、部分実行のアルゴリズムを融合する。そして、各部分木に対応する目的プログラムを、一つの部分実行範囲と見なすことにより、目的プログラムの部分実行を実現させる。

提案手法の概念図を図2に示す。



(a)解析木の部分木　　(b)目的プログラム

図2　部分木と目的プログラムの対応
(in、out：部分実行範囲内の使用変数の値)

図2のように、解析木の部分木ごとに生成される目的プログラムを、部分実行するときの各部分実行範囲とみなす。そして、各部分木内で使用される変数名の集合をコンパイル時に検索し、保存する。その様子を表1に示す。

表1　各部分実行範囲内で使用される変数保存の例

部分実行範囲	使用変数	in	out
1	a, b, c	…	…
2	d, e, f	…	…

表1のような変数保存に使用する表を、以後、Look Aside Tableと呼ぶことにする。

次に、一回目の目的プログラム実行時に、各部分実行範囲内の処理前後の変数値（in、out）をLook Aside Tableへ保存する。そして、差分コンパイル後の再実行時に、Look Aside Tableの参照を随時行う。変数集合の値に変化がある場合は、その部分実行範囲を再実行する必要がある。しかし、変化のない場合は、再実行が不要となる。Look Aside Tableの更新は、目的プログラムの実行ごとに毎回行う。

本研究では、基本的な構文として、代入文、制御文（while文、if文、call文、return文）に対する部分実行を扱う。

代入文に対する部分実行では、Look Aside Tableへの保存、参照を各代入文について行う必要がある。しかし、この方法では、メモリ容量が増大し、且つ、処理時間が劣化してしまう。そこで、コンパイル時に、代入文が連続する場合は、そのまとまりを一つの部分実行範囲と判断し、まとまりごとの保存、参照を行う。その様子を図3に示す。

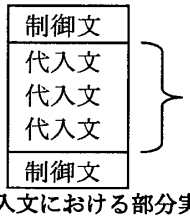


図3　連続する代入文における部分実行範囲の区切り

制御文に対する部分実行では、制御文の中身を修正する場合、部分実行の影響範囲が変更される可能性がある。よって、制御文以降全ての再実行を想定する必要があり、その影響範囲を如何にして判断するかが重要となる。

call文、return文によるサブルーチン単位の部分実行では、Look Aside Tableへの保存、参照を引数と戻り値で行う。サブルーチンの流れを図4に示す。

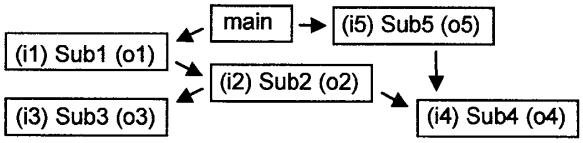


図4　サブルーチンの流れ
(i：引数、o：戻り値)

図4において、Sub4の中身が修正された場合を想定する。Sub4の再実行後、戻り値o4に変化がある場合、上位のサブルーチンSub2、Sub5及び、Sub1、mainのcall命令以降の再実行を想定する必要がある。しかし、その再実行の過程で、例えば、Sub2の戻り値o2に変化がなければ、それ以降のSub1、mainの再実行が不要となる。また、Sub1の中身が修正された場合を想定する。Sub1の再実行中、呼び出されるSub2の引数i2に変化がある場合、下位のサブルーチンSub2、Sub4の再実行を想定する必要がある。しかし、Sub2の引数i2に変化がなければ、Sub2、及び、それ以降のSub4の再実行が不要となる。ソースを用いた簡単な例を図5、図6に示す。

Incrementation Execution by means of Incremental Compiling: Takeru Inoue (Hosei Univ.)

```
Sub4(a){ x=1; if(x>1) a++; return a; }
Sub2(a){ b=Sub4(a); if(b<3) b++; return b; }
Sub1(a){ b=1; c=Sub2(b); …; }
```

図5　サブルーチン単位の部分実行の例1

図5において、一回目の実行時のSub2、Sub4の戻り値は1である。変数xの値を2に修正した場合、再実行時にSub4の戻り値が2に変化する。そのためSub2のcall命令以降を再実行する必要がある。しかし、Sub2の戻り値には変化がないので、Sub1のcall命令以降の再実行が不要となる。

```
Sub2(a){ …; }
Sub1(a){ x=1; if(x<3) a++; b=Sub2(a); …; }
```

図6　サブルーチン単位の部分実行の例2

図6において、一回目の実行時のSub2の引数は1である。変数xの値を2に修正したとしても、Sub2の引数に変化がないので、Sub2の再実行が不要となる。

while文、if文に対する部分実行では、使用変数の処理前の値をサブルーチンの引数、処理後の値を戻り値とみなす。そして、サブルーチンと同様の部分実行を行う。その様子を図7に示す。

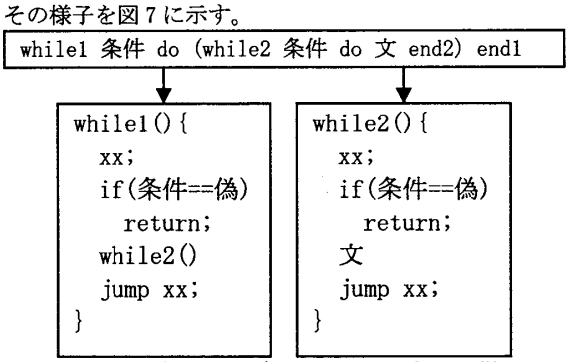


図7　制御文をサブルーチンとみなした様子

ソースを用いた簡単な例を図8に示す。

```
while(i<10){
  a=0; j=0; x=1; …;
  while(j<10){ a++; j++; }
  …; i++;
}
```

図8　制御文に対する部分実行の例

図8において、変数xの値を修正した場合、外側のwhile文については、全ての再実行を想定する必要がある。しかし、内側のwhile文については、Look Aside Tableに変化がないので、再実行が不要となる。

3. 実験

部分木ごとの差分コンパイルを行う差分コンパイラと、PL/0の解釈子、及び、提案手法による部分実行を行う解釈子の実装を行い、以下の実験を行う。尚、本実験では、Modula-2言語のソースを使用する。

実験1　差分コンパイラと、部分実行を行う解釈子を用いて、以下の実行についての処理時間を測定する。

- 各命令に比べて、極端に処理時間の長い繰り返しループを一つ含んだ目的プログラムの全実行
- 同ソースにおいて、繰り返しループに影響を与えない箇所を一部修正した場合に、差分コンパイルにより得られた目的プログラムの部分実行

実験2　PL/0の解釈子を用いて、以下の実行についての処理時間を測定する。

- 実験1における部分実行範囲と同じ目的プログラムの実行。

結果　実装したアプリケーションの実行例を図9に示す。

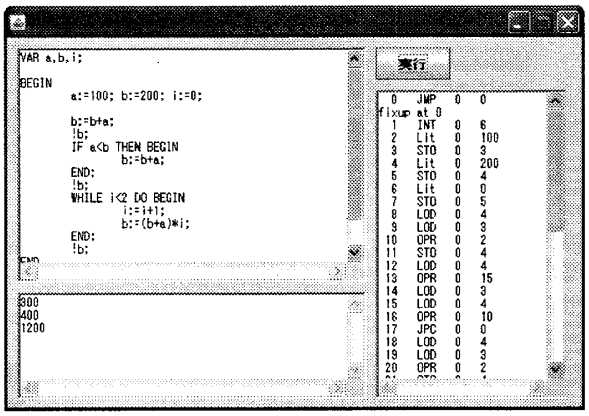


図9　実装したアプリケーション

また、実験1の測定結果を表2、実験2の測定結果を表3に示す。

表2　部分実行による処理時間短縮例

処理内容	処理時間［μs］
全処理の実行	26860
部分実行	2028

表3　PL/0解釈子実行の例

処理内容	処理時間［μs］
PL/0の解釈子による実行	409

4. 考察

表1を見ると、全処理の実行時間26860[μs]に対し、部分実行時間は、2028[μs]であることが確認できる。よって、繰り返し文等により長時間の実行を要するプログラムの場合、その繰り返しループ外のみの再実行が可能であることが分かる。しかし、表2を見ると、PL/0の解釈子による目的プログラム実行時間が409[μs]であることが確認できる。よって、実行時間の短いプログラムにおいて、同じ量の目的プログラムを実行する場合、PL/0の解釈子に比べ、提案手法による実行では、395[%]処理時間が劣化していることが分かる。これは、Look Aside Tableの参照による処理時間の劣化である。

5. 結論

本研究では、修正した範囲のみの実行、実行結果の更新が可能な部分実行と、部分木ごとの差分コンパイルを行う差分コンパイラのアルゴリズムを融合させた。そのアルゴリズムにより、目的プログラムの変更部、及び、その関連部分のみに限定して部分実行する手法を提案した。また、本提案手法は、実行時間の短いプログラムの修正作業においては効果がないが、実行時間の長いプログラムの修正作業において有益であることを実証した。そして、差分コンパイル後の目的プログラム実行作業において、全ての目的プログラムを実行する必要がなくなり、差分処理時間の短縮を可能とした。

参考文献

[1] Wagner, T.A. and Graham, S.L.: Efficient and Flexible Incremental Parsing, Full text Pdf (2.29 MB), Source ACM Transactions on Programming Languages and Systems(TOPLAS), pp.980-1013 (1998)