組み込み向け低レベル言語 CbC の GCC による実装

与儀 健人　　河野 真治

1　概要

当研究室では Continuation based C(以下 CbC) という言語を提案しており、そのコンパイルにはこれまで Micro-C をベースにした独自のコンパイラを使用していた。また、以前の論文で GCC の Tail call optimization を用いて GCC 上に実装が可能である事を示した。ここでは GCC 上に実際に CbC 言語の実装し、その評価を行った。この実装はアーキテクチャに依存しないので、GCC が対応する全てのアーキテクチャ上で CbC が動く事になるはずであるが、若干の問題があり、その点に付いても考察を行う。

2　CbC について

Continuation based C は当研究室が提案するアセンブラよりも上位で C よりも下位な記述言語である [2]。C の仕様からループ制御や関数コールを取り除き、継続 (goto) や コードセグメントを導入している。これによりスタックの操作やループ、関数呼び出しなどのより低レベルでの最適化を、ソースコードレベルで行うことができる。

図 1 はコードセグメント同士の関係を表したものである。コードセグメント `start` は実行を終えると goto によって別のコードセグメント `A` もしくは `B` に実行を継続する。また、`A` から `B`, 再び `A` の用に継続を繰り返すことも可能だ。このように、コードセグメントから goto を用いて別のコードセグメントへ飛ぶ構成はオートマトンと似た構造になっていることがわかる。

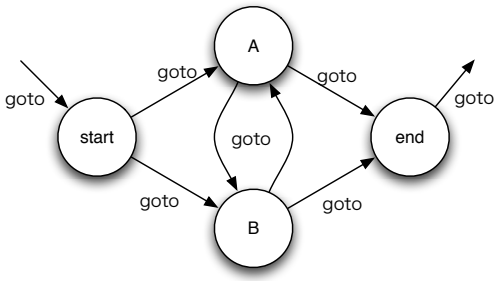

図 1　コードセグメント間の “継続”

これらの特徴から、CbC は自身でスケジューラの記述ができ、それにより並列処理や逐次処理をスムースに繋げることが出来る。また、OperatingSystem の記述やハードウェアの記述に向いている。

以下では実装に必要な CbC の構文、コードセグメントの定義と継続 (goto) について説明する。

コードセグメント は CbC における最も基本的な処理単位である。コードセグメントを定義する構文は通常の関数と同じであるが、型は “__code” となる。ただし、コードセグメントは関数のようにリターンすることはないので、これはコードセグメントであることを示すフラグの様なものである。

コードセグメントの処理内容も通常の関数と同じように定義されるが、C と違いコードセグメントでは for や while, return などの構文は存在しない。ループ等の制御は自分自身への再帰的な継続によって実現されることになる。

継続 (goto) はコードセグメント間の移動を表す。構文としては goto をつかっているが C における label への goto とは違い、goto の後ろに関数呼び出しの様な形をとる。例として、あるコードセグメント `cs` への継続は `goto cs(10, "test");` となる。これにより、cs に対して引数 `10` と`"test"`を渡すことができる。ただし関数コールとは違い、継続ではコールスタックの拡張を行わない。代わりに goto を発行したコードセグメントの持つスタック自体に次のコードセグメントの引数を書き込むことになる。また、return アドレスの push なども行わない。

2.1 CbC の例文

以上の 2 つの構文を使った例題をリスト 1 に示す。

リスト 1 CbC 例

```
__code while_cond(int total, int count){
    if ( count <= 100 ){
        goto while_process(total,count);
    }else{
        goto while_end(total);
    }
}
__code while_process(int total,
                     int count){
    /* some processes */
    goto while_cond(total, count);
}
__code while_end(int total){
    goto cs_exit(0);
}
```

これは単純なループ構造である。まず `while_cond` が実行されると、そこでは条件判定により `while_process` か `while_end` に継続する。`while_process` では処理が終了すると再び `while_cond` に継続することでループが形成される。このように CbC では for や while を使用せずコードセグメントから同じコードセグメントへ継続する形でループが表現される。

3 GCC の構成

今回の実装では GCC のソースコードを修正することになる。また、GCC の最適化処理の一つである Tail call がその実装に深く関わってくる [1]。この章では GCC の基本的な動作について簡単に説明する。

3.1 GCC の基本構造

GCC は pass と呼ばれる一連の処理の中でソースコードをアセンブリに変換する。以下ではその pass の中でも重要なものをその実行順に説明する。

parsing パーサによってソースコードを解析する。解析した結果は Generic Tree と呼ばれる tree 構造の構造体に格納される。

gimplification Generic Tree をもとにこれを GIMPLE に変換する。

GIMPLE optimization GIMPLE に対して最適化を行う。

RTL generation GIMPLE をもとに RTL を生成する。

RTL optimization RTL に対して最適化を行う。

Output assembly RTL をもとにターゲットマシンのアセンブリに変換する。

これらの処理は図 2 のように表される。 各 pass は通

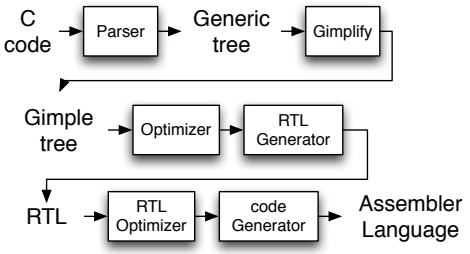

図 2 GCC の pass

常は各々の関数毎に行われるものだが、inline 関数の処理や、関数間での最適化を行う場合には一つのソースファイル毎に行われる。

3.2 Tail call elimination

最適化のひとつである “Tail call elimination” †1は、“関数の return の前に別の関数呼び出しがある場合は

†1 Tail call Optimization と呼ばれたり、単に Tail call と呼ばれたり、呼称はさまざまである。

call 命令でなく jump 命令が使える” というアイデアを元に設計されている。この最適化は本研究における CbC コンパイラの実装に深く関わってくるので、以下で詳しく説明する。

3.2.1 Tail call の概要

まず main 関数から関数 A を呼び出していて、関数 A の最後の処理 (return 直前) では次に関数 B を呼び出している状況を考える (図 3 参照)。このあと関数 B の処理が終了すると、ret 命令により一旦関数 A に戻ってきて、そこで再び ret 命令をつかって main に戻ることになる。“Tail call elimination” ではこの B から A に戻る無駄な処理を低減する。この様子を図 3 に示したので参考にしていただきたい。

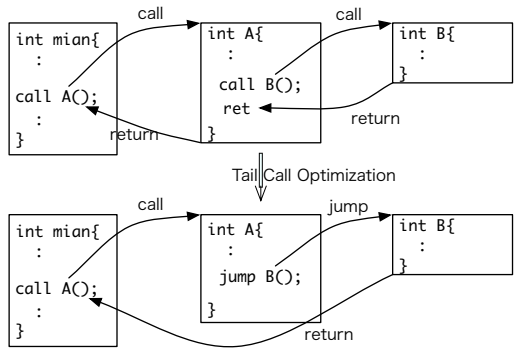

図 3　Tail call elimination の例

次に “Tail call elimination” によって、アセンブリレベルでどのようにコードが変わるのか、スタックの変化も交えて見てみる。この例では最も一般的に使われている i386 形式のアセンブラを使用している。

図 3 と同じように呼び出される関数 main, A, B をリスト 2 の様に定義する。

リスト 2　関数 main A B の例

```
void B(int A, int A, int C){
    /* what do you do?  */
    return ;
}
void A(int a, int b, int c, int d){
    /* some processes..  */
    return B(a, b, c+d);
}
int main(int argc, char **argv){
    A(10, 20, 30, 40);
    return 0;
}
```

これを通常通り、“Tail call elimination” を使用せずにコンパイルすると次のリスト 3 のようなコードが出力される。ここでは Tailcall 最適化が影響をあたえる関数 A のみをしめした。また、出力アーキテクチャは i386 である。

リスト 3　関数 A のコンパイル結果 (Tail call なし)

```
A:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $24, %esp
    movl    20(%ebp), %eax
    addl    16(%ebp), %eax
    movl    %eax, 8(%esp)
    movl    12(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    8(%ebp), %eax
    movl    %eax, (%esp)
    call    B
    leave
    ret
    .size   A, .-A
```

これを見ても分かる通り、Tail call をしない場合は A のスタック領域の上に B のスタック領域が確保され、B が終了するとそれが破棄される形になる。

次に Tail call elimination が行われた場合のコンパイル結果をリスト 4 に示す。

リスト 4　Tail call elimination の行われた関数 A

```
A:
    pushl   %ebp
    movl    %esp, %ebp
    movl    20(%ebp), %eax
    addl    %eax, 16(%ebp)
    popl    %ebp
    jmp     B
    .size   A, .-A
```

20(%ebp) は変数 d、16(%ebp) は変数 c を表している。ここでは B のためにスタック領域は確保せず、かわりに A のスタック領域に B のための引数を上書きしていることが分かる。ただし、変数 a と b は書き込む位置も値も変わらないので触れられていない。また、call 命令を使わずに jmp で B に飛んでいる (そのためリターンアドレスも確保してない)。これによ

り、B 側では ret 命令を発効すると A に戻らず、A の呼び出し側に直接戻ることになる。

3.2.2 Tail call 時のスタック

このときのスタックの様子を図 4 に表した。 図 4

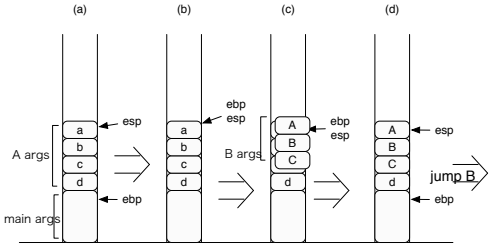

図 4 関数 A から B を呼び出す時のスタックの様子

の各ステップは次のような状態を表している。

(a) main から A が呼ばれた直後の状態。esp は引数のトップをさしてるが、ebp は main の引数をさしたまま

(b) ebp を esp に合わせる。通常は ebp のオフセットから引数のアドレスを指定する。

(c) A 自身のスタックフレームに B 用の引数をつめる。

(d) ebp を元に戻す。その後関数 B に jump。

(a),(b) は関数 A の初期化処理、 (c),(d) は関数 B の呼び出し処理である。通常は関数呼び出しの際は A のスタックフレームの上に新たに作るはずである。しかし、関数 A の Tail call elimination 後のコードを見ても分かる通り、無駄な処理が少なくなっていることが分かる。これが Tail call elimination における最適化の主な効果である。最大の効果が得られるのは、caller 関数が持っている引数を callee 関数に直接渡す場合である。この時はスタック操作は全く必要なく、単に jump 命令のみになる。

3.3 Tail call の条件

Tail call が可能かどうかの条件についてここで考察する。

まず最初の条件として、“関数コールが return の直前にある” ということは自明だろう。また、これに関連して “関数の返す型が caller と callee で一致している” ことが必要となる。

図 4 の (c) にて callee 関数 B のための引数をスタックに上書きしているが、この領域は A のためのスタックフレームであることは説明した。ここでもし B の引数が 5 つ以上あったらどうなるだろうか? 図を見て分かる通り、main のスタックまで書きつぶすことになってしまう。このことから “caller 側の引数サイズが callee 側の引数サイズより大きいもしくは等しい” という条件が必要だと分かる。

最後に callee 用の引数を格納する順番が問題になる。通常、引数は関数定義の右から順にスタックにつめられる。例えば図 2 のコードにおいて、A から B の呼び出しが `B(c, b, c+d)` となっていたらどうだろうか? 最初に `c+d` の書き込みによって変数 c は上書きされてしまう。そのため、最後に書き込む引数 c は上書きされた c+d が使われ、実行結果はまったく違うものになってしまうだろう。よって、“書き込んだ引数が、その後書き込む引数を上書きしてはならない” という条件も必要となる。

他にも細かな条件はあるが、以上の考察より以下の 4 つの条件が明らかになった。

i) 関数コールが return の直前にある

ii) 関数の返す型が caller と callee で一致している

iii) caller 側の引数サイズが callee 側の引数サイズより大きいもしくは等しい

iv) 書き込んだ引数が、その後書き込む引数を上書きしてはならない

CbC コンパイル機能の実装の際にはこれらの条件をパスさせる必要がある。

4 実装

次に、実際の実装方法を簡単に説明する。今回実装したソースコードは SourceForge 上の以下の URL にて公開されている。

`http://sourceforge.jp/projects/cbc/`

実装に置ける最大の問題は goto 文でのコードセグメントへの jump の際にスタックフレームをどう扱うかということである。2 章にて説明した通り、CbC ではコードセグメントへジャンプした後は元のコード

セグメントには戻らない。ゆえに通常の関数コールと違い、スッタクを積む必要が無い。この特性が 3.2.1 で説明した Tail call の性質と似ていることを利用し、“コードセグメントへの goto はすべて Tail call に置き換える” という実装を行う。

4.1 Tail call 条件のパス

上記のような実装を行う上で、3.3 で説明したように、Tail call の条件を満足させる必要がある。

条件の i は簡単に実装できる。例えば次のような継続文を考える。

```
goto cs(10, "test");
```

この文を構文解析する際に次のような形に置き換えることになる。

```
cs(10, "test");
return;
```

これにより Generic Tree の段階では単純な関数呼び出しとなるが、Tail call のフラグをたてることにより、RTL 生成時には jump 命令に置き換えられる。

次に条件の ii だが、これは “__code” を型に持つものはすべて void 型に置き換えることで対応できる。すなわちコードセグメントの継続は void 型関数への Tail call となる。

条件 iii が最大の問題となる。もしより大きい引数サイズのコードセグメントに Tail call すると、直前の呼び出した関数スタックを書きつぶしてしまう。最悪 Segmentation fault となる。今回はこの解決方法として、すべてのコードセグメントでは引数スタックを一定量とした。ゆえにこのサイズ以上の引数を持つことはできないが、これは通常のプログラミングでは問題にならない程度に大きい値にしている。また、継続の際にはスタック拡張を行わないので、この一定量が大きくなってもとくに実行速度等に影響はない。

最後に条件 iv だが、これは上書きされうる引数がある場合には直前にそれを計算し一時変数に代入するように修正した。

以上の内容に基づいて修正したソースコードの大半は RTL を生成部となる。

5 環境付き継続に関する考察

また前説までにその実装方法を説明した。しかしまだ実装されていない構文があるので、その実装方法に関してここで考察する。

5.1 環境付き継続の概念

環境付き継続は以下に示すように、継続文の後ろに “環境” の値を明示したものである。

```
goto cs(10, 20), env;
```

この構文を使用することでスタックフレームを別の (環境 env が示す) 領域に切り替えたうえで継続を行うことができる。例としてリスト 5 の様なコードセグメントが考えられる。

リスト 5 環境付き継続 例

```
__code envSwitch(int a, int b, double c)
{
    void *stack;
    if ((stack=malloc(STACKSIZE))==NULL)
        goto error(no);

    goto envCheck(10, 20), stack+OFFSET;
}
```

このコードセグメント envSwitch ではスタック領域を malloc で取得した領域に切り替えた上で envCheck に継続する[†2]。

また、この構文を使ってコードセグメントを呼び出した関数に戻ることも可能となる。それにはこの関数側でも若干の修正が必要で、`__return`, `__environment` という二つの定義済み変数を使用する。これらはそれぞれコードセグメントから関数への戻り先とその時のスタックフレームの位置を表している。これらをリスト 6 の様に用いることで `env_func` を呼び出した関数まで戻ることができる。また、`env_code` 環境付き goto で与えている引数は `env_func` の戻り値として扱われる。

リスト 6 呼び出し元の関数への goto 例

```
__code env_code(int a, int b,
```

†2 この構文を使用し、複数のスタックを交互に切り替える等の処理を行うことでタブロー法などの検証を行うのが CbC の目的でもある。

```
                void *env, void *_ret)
{
    __code (*ret)(int, int, int);
    ret = _ret;
    goto ret(20), env;
}
int env_func(int a, int b, double c)
{
    goto env_code(20, 30,
        __environment, __return);
    /* unreachable */
}
```

5.2 実装方法に関する考察

環境付き継続の実装方法について考察を行う。環境付き継続では通常の関数コールや継続と違い、スタックフレームを変更するため、現在のスタックフレームの位置をさすレジスタを変更しなければならない。理想的なリスト 5 のコンパイル結果は以下のようになる。

リスト 7　環境付き継続 出力例

```
movl    $env, $eax
movl    $10, 8(%eax)
movl    $20, 12(%eax)
movl    %eax, %ebp
jmp     envCheck
```

この結果なら新たなスタックフレーム env の先に引数を格納し、かつ%ebp レジスタを置き換えた上でコードセグメントに継続している。

しかし、現在の Tail call を用いた仕様ではいくつかの問題点がある。一つは “Tail call は jmp 命令の直前に必ず%ebp を変更する” ということである。さらに “環境” の値から引数を格納する位置までのオフセットを計算する必要がある。これは GCC では RTL として virtual_incoming_args_rtx や hard_frame_pointer_rtx などの固定レジスタ値が用意されてるので、これを用いて計算できるだろう。最後に、関数の呼び出し側に戻る時の__return 変数の定義が必要となる。この変数はただ値をつくるだけでなく、この変数が使われている場合には特殊なコードを生成する必要がある。

6 評価

今回、環境付き継続は実装には至らなかったが、残りの部分は実装完了した。そこでベンチマークテストを行い、Micro-C との比較を行った。

今回ベンチマークに使用したプログラムはこれまでも Micro-C の測定に使われていたテストルーチンで、普通の C のソースをプログラムで CbC に変換したものである。引数に 1 を入れるとそれが変換された CbC のコード、引数 2,3 では変換されたコードを手動で Micro-C 用に最適化したコードが実行される。また、評価は ia32 アーキテクチャの Fedora 上で行った。一番結果の良い引数 2 の場合のコードセグメントをリスト 8 に示す。

リスト 8　bench

```
__code  f2(int i,char *sp) {
    int k,j;
    k = 3+i;
    goto g2(i,k,i+3,sp);
}
__code g2(int i,int k,int j,char *sp){
    j = j+4;
    goto h2(i,k+4+j,sp);
}
__code h2_1(int i,int k,int j,char *sp){
    goto main_return2(i+j,sp);
}
__code h2(int i,int k,char *sp) {
    goto h2_1(i,k,i+4,sp);
}
```

このベンチマークでは CbC の継続と計算を交互に行っている。測定結果は表 1 に示される。

	./conv1 1	./conv1 2	./conv1 3
Micro-C	8.97	2.19	2.73
GCC	4.87	3.08	3.65
GCC (+omit)	4.20	2.25	2.76
GCC (+fast)	3.44	1.76	2.34

表 1　Micro-C, GCC の実行速度比較 (単位 秒)

通常の Tail call elimination のみを有効にした場合の結果が 2 行目、その他は次節で説明するオプションを付加したものである。見てのとおり、手動で最適化された引数 2,3 の場合はオプションを加えなければ

Micro-C の速度に及ばなかった。次節ではこの点について考察する。

6.1 出力コード

先ほどのリスト 8 のコードセグメント g2 のみを Micro-C でコンパイルした結果をリスト 9, GCC のオプション無しによるコンパイル結果をリスト 10 に示す。

リスト 9 Micro-C による出力コード

```
f2:
    lea -_44(%ebp),%esp
    movl $3,%eax
    addl %esi,%eax
    movl %eax,-28(%ebp)
    movl %edi,-32(%ebp)
    movl -28(%ebp),%edi
    movl %esi,%eax
    addl $3,%eax
    movl %eax,-28(%ebp)
    jmp g2
```

リスト 10 GCC による出力コード

```
f2:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %eax
    movl    12(%ebp), %ecx
    leal    3(%eax), %edx
    movl    %edx, 12(%ebp)
    movl    %edx, 16(%ebp)
    movl    %ecx, 20(%ebp)
    popl    %ebp
    jmp     g2
```

このとおり出力コードは 10 命令と、行数にはあまり差が無い。(他の segment も同様である) しかし GCC の出力においては無駄なコードが混じっていることがわかるだろう。`pushl%ebp` と `popl %ebp` である。すべてのコードセグメントにおいてこれと同じ命令が出てしまっているが、これは Tailcall を無理矢理適用したために出てきたコードである。

このような関数の最初と最後にある無駄なフレームポインタの push を抑制するオプションが-fomit-frame-pointer である。このオプションを付加するとリスト 11

リスト 11 GCC による出力コード

```
f2:
    movl    4(%esp), %eax
    movl    8(%esp), %ecx
    leal    3(%eax), %edx
    movl    %edx, 8(%esp)
    movl    %edx, 12(%esp)
    movl    %ecx, 16(%esp)
    jmp     g2
```

これによって一気に 3 命令減った。ベンチマークは表 1 の 3 行目、“GCC (+omit)” である。しかし、(コードセグメントにもよるが)3/10 命令減ったにもかかわらず Micro-C との速度差がほとんど無い。

リスト 9 をみると Micro-C では引数の格納にレジスタ%edi と%esi を用いる分、高速なコードを生成出来ていることが分かる。この違いが命令数の差を埋めている。GCC でも引数をレジスタに詰めることができる fastcall 属性がある。-fomit-frame-pointer に加えて fastcall を付加した結果をリスト 12 に示す。

リスト 12 GCC による出力コード

```
f2:
    movl    %edx, %eax
    leal    3(%ecx), %edx
    movl    %edx, 4(%esp)
    movl    %eax, 8(%esp)
    jmp     g2
```

命令数はさらに 2 命令減り、またメモリへのアクセスが減ったためベンチマーク結果 (表 1GCC (+fast)) も大幅に改善した。

この評価結果から、GCC の最適化オプションを用いることで CbC コードのさらなる高速化が可能であることが示された。また、使用したベンチマークプログラムは C のコードをプログラムで CbC に変換したものだが、これを C のままコンパイルすると最適化をかけても約 3.3 秒かかる。このように不要なスタック操作を減らすことによって、C 言語のみでは不可能な手動による最適化が CbC の利点としてあげられる。

7 まとめ

本研究の実装により、GCC を使って CbC のソースコードをコンパイルすることができるようになった。その結果、これまでの Micro-C ベースのコンパ

イラではできなかった最適化を GCC 上で実行できるようになり、CbC プログラムをより高速化することに成功した。また、環境付き継続の実装方法に関して考察を行いその問題点を洗い出した。

今後は 5 で説明した様に環境付き継続の実装が課題となる。また、SPU アーキテクチャに GCC が対応してないという問題もある。今回の実装の目的の一つとして PS3 上で CbC を動かしたいということがあったので、この問題については PS3SDK とのマージも一つの手法として考えている。これらに加えて、GCC はすでに C++や Objective-C のコンパイルが可能である。これを活かし、CbC++, もしくは Objective-CbC といった既存の言語と CbC を組み合わせた言語に付いても今後実装していく。

参考文献

[1] 河野真治. “継続を基本とした言語 CbC の gcc 上の実装”. 日本ソフトウェア科学会第 19 回大会論文集, Sep, 2002.

[2] 河野真治. “継続を持つ C の下位言語によるシステム記述”. 日本ソフトウェア科学会第 17 回大会論文集, Sep, 2000.

[3] Simon Peyton Jones, Thomas Nordin, and Dino Oliva, “C–: a portable assembly language”. Implementing Functional Languages, 1997.

[4] GNU Project - Free Software Foundation, GCC internal manual. `http://gcc.gnu.org/onlinedocs/gccint/`.