Continuation based C の GCC 4.6 上の実装について

大 城 信 康† 河 野 真 治†

GCC-4.6 をベースとした CbC コンパイラの実装を行った. CbC のコンパイラは GCC-4.2 ベースのコンパイラが 2008 年に開発されており, 以来 GCC のアップデートにあわせて CbC のコンパイラもアップデートが行われてきた. 今回は GCC-4.6 への実装を行った. 本論文では GCC-4.6 への CbC の具体的な実装について述べる。

The implementation of Continuation based C Compiler on GCC 4.6

Nobuyasu Oshiro† and Shinji Kono†

We implemented Continuation based C Compiler on GCC-4.6. CbC Compiler on GCC-4.2 was developed on 2008. Since then we kept to update it. In this paper, we introduce implemented Continuation based C Compiler on GCC-4.6.

1. 歴史的経緯

当研究室ではコードセグメント (Code Segment) 単位で記述するプログラミング言語 Continuation based C (以下 CbC) を開発している. コードセグメントは並列実行の単位として使うことができ, プログラムの正しさを示す単位としても使用することができる. これにより, Many Core での並列実行を高い性能と高い信頼性で実現することができると考えている.

CbC のコンパイルには元々 Micoro-C 版の独自のコンパイラを用いていたが, 2008 年の研究において GCC-4.2 ベースの CbC コンパイラが開発され, 2010 年には GCC-4.4 へとアップデートが行われた. GCC への実装により,GCC の最適化やデバッガの機能を使うことができより実用的な CbC プログラミングが行えるようになった. だが, GCC をベースとした CbC のコンパイラ (以下 CbC-GCC) は, GCC のアップデートに合わせて変更する必要がある. 本研究では, GCC-4.5 をベースとしていた CbC-GCC を GCC-4.6 へのアップデートを行い, Intel64 に対応するとともに, CbC の拡張を行う.

2. Continuation based C (CbC)

CbC のプログラムはコードセグメント毎に記述され, コード間を goto(軽量継続) により処理を移る. 構文は C と同じであるが, ループ制御や関数コールが取り除かれる.

2.1 継続 (goto)

コードセグメントの記述は C の関数の構文と同じで, 型に“ `code` ”を使うことで宣言できる. コードセグメントへの移動は“ goto ”の後にコードセグメント名と引数を並べて記述することで行える. 図 1 はコードセグメント間の処理の流れを表している.

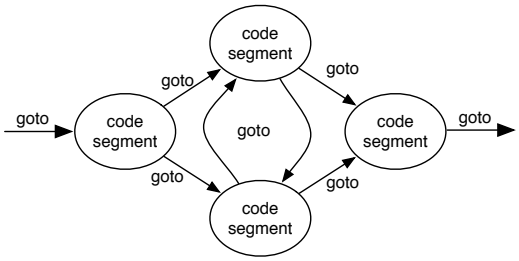

図 1 コードセグメント間の継続 (goto)

2.2 コードセグメント (code segment)

コードセグメントは C の関数と違って返り値を持たず, 処理が終われば次のコードセグメントへと処理を移る. C において関数呼び出しを繰り返し行う場合, 呼び出された関数の引数の数だけスタックに値が積まれていく. だが, 返り値を持たないコードセグメントではスタックに値を積んでいく必要な無く, スタックは変更されない.

軽量継続により並列化, ループ制御, 関数コールと

† 琉球大学
University of the Ryukyu

```
__code print_factorial(int prod)
{
  printf("factorial = %d\n",prod);
  exit(0);
}
__code factorial0(int prod, int x)
{
  if ( x >= 1) {
    goto factorial0(prod*x, x-1);
  }else{
    goto print_factorial(prod);
  }
}
__code factorial(int x)
{
  goto factorial0(1, x);
}
```

図 2 階上を計算する CbC プログラムの例

スタックの操作を意識した最適化がソースコードレベルで行えるようになる.

図**??**は CbC で書いたプログラムの例である. 与えられた数 x の階上を計算して出力するプログラムとなっている.

3. GCC の 3 つの内部表現

GCC-4.6 への実装の前に,GCC で扱われる 3 つの内部表現について触れておく.

3.1 3 つの内部表現

GCC は内部で Generic Tree, GIMPLE, RTL という 3 つの内部表現を扱う. それぞれが読み込んだソースコードは Generic Tree, GIMPLE, RTL の順に変換されていき, 最後にアセンブラ言語へと出力される. 図 3 は GCC がソースコードを読み込みアセンブラ言語出力までの流れを表した図である.

3.1.1 Generic Tree

ソースコードより読み込んだ関数の情報を木構造で表したものが Generic Tree となる. 関数の返り値, 引数, 変数の型, 条件式とプログラムの処理全てが木構造で表される. CbC の実装では parse の部分からこの Generic Tree 生成の部分に手が加わっている.

3.1.2 GIMPLE

Generic Tree で表現されたデータは GIMPLE というデータ構造に変換される. GIMPLE も Generic Tree と同じ構文木だが、より制約がかかった状態で

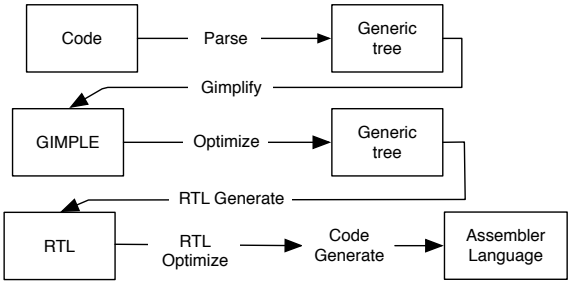

図 3 GCC によるコンパイルの一連の流れ

作成された構文木となる. 制約は「1 つの枝に 4 つ以上の子を持たせない」等といったもので, GIMPLE へと変換されたデータは Generic Tree より簡単な命令で表されることになり最適化がかけやすくなる. CbC の実装では特に修正は加えていない.

3.1.3 Register Transfer Language (RTL)

構文木 GIMPLE は解析が行われた後 RTL へと変換される. RTL はレジスタの割り当てといった低レベルの表現で, アセンブラとほぼ同じ表現を行うことができる. プログラム内部では RTL も木構造で表される.

CbC における継続は, この RTL への変換で行われる最適化の 1 つ Tail Call Elimination が重要となってくる.

4. GCC-4.6 への実装

前節までで CbC の基本仕様と GCC でのアセンブラ出力までの流れを確認した. ここからは GCC-4.6 への実装について述べていく.

4.1 Tail Call Elimination

CbC の継続の実装には GCC の最適化の 1 つ, Tail Call Elimination (末尾除去) を強制することで実装する. これにより, コードセグメント間の移動を, call ではなく jmp 命令で実現する. 図 4 は Tail Call Elimination が行われた際のプログラムの処理を表している.

funcB は jmp 命令で funcC を呼び出す. funcC は, 返り値を funcB ではなく funcA へと返すことになる.

4.1.1 expand_call

ある関数が Tail Call Elimination を行えるかどうかは `expand_call` 関数で判断される. `expand_call` 関数内でチェックされる Tail Call Elimination が行える条件は以下になる.

- caller 側と callee 側の戻値の型が一致している.
- 関数呼び出しがリターンの直前に行われている.

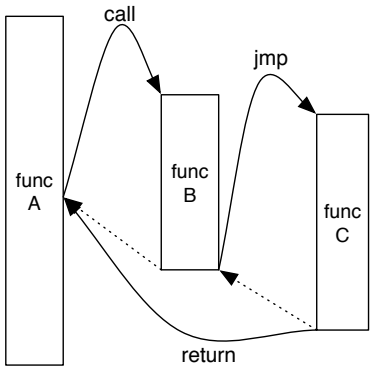

図 4 Tail Call Elimination

- 呼出先関数の引数に用いられるスタックサイズが呼出元関数のそれより少ない.
- 引数の並びのコピーに上書きがない.

CbC の実装では上記の条件を, 以下の様にして解決させている.

- コードセグメントは void 型で統一する
- C の関数からコードセグメントに goto する場合は返す値の型チェックを行わない.
- goto の直後に retrun を置く.
- スタックサイズは関数宣言時に決まったサイズにする.
- 引数は一旦, 一時変数にコピーして重なりがないようにする.

スタックサイズを決め打ちで行うことで, ベースポインタを変えずにスタックを扱うことができる. これも CbC の 1 つの特徴である. 図 5 はコードセグメントの継続の際にスタックに積まれる引数を表している.

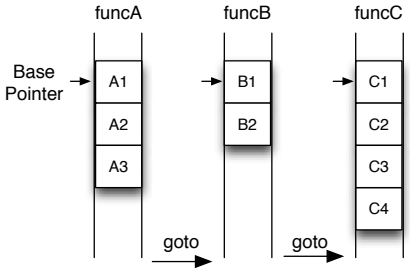

図 5 継続による引数のスタック格納の様子

expand_call 関数の中で try_tail_call フラグ

4.2 引数渡し

通常コードセグメントの継続において, 引数は C の関数と同じスタックを用いて渡される. GCC には引数渡しをスタックではなくレジスタを用いて行う機能として fastcall がある. fastcall を用いてコードセグメントを宣言することで, レジスタを用いた速度の向上を図る.

4.2.1 fastcall

コードセグメントの引数渡しを fastcall によりできるだけレジスタを用いて行うようにする. C において fastcall を用いる場合は関数にキーワード “ __attribute__ ((fastcall)) ” をつけて行う. だが, コードセグメントを全てこのキーワードをつけて宣言することは実用できではない. そこで, コードセグメントで宣言された場合,fastcall が自動で付くように実装を行う. 図 6 はコードセグメントに fastcall 属性を付与しているソースである.

```
 1 |  case RID_CbC_CODE:
 2 |    if (!typespec_ok)
 3 |      goto out;
 4 |    attrs_ok = true;
 5 |    seen_type = true;
 6 |    if (c_dialect_objc ())
 7 |      parser->objc_need_raw_identifier = true;
 8 |    t.kind = ctsk_resword;
 9 |    t.spec = c_parser_peek_token (parser)->value;
10 |    declspecs_add_type (loc, specs, t);
11 |
12 |    if(!TARGET_64BIT) {
13 |      attrs = build_tree_list (get_identifier("fastcall"), NULL_TREE);
14 |      declspecs_add_attrs(specs, attrs);
15 |    }
16 |
17 |    c_parser_consume_token (parser);
18 |    break;
```

図 6 コードセグメントへの fastcall 属性付与

if 文で条件を決めているのは,64 bit の場合 fastcall が標準で行われ, fastcall 属性を付けると warning を出すからである.

5. 評　　価

今回実装を行った GCC-4.6 ベースのコンパイラを GCC-4.4 ベース, Micro-C コンパイラとそれぞれ比較を行った. 比較を行うのはクイックソートのプログラムである. クイックソートは再帰的にプログラムされる為 CbC に向いているプログラムだと言える. 比較を行うのは以下のアーキテクチャと OS になる.

- x86/Linux
- x86/OS X

また, 比較を行うプログラムは最適化 (-O0 オプション) を行わないものと, 速度最適化 (-O2 -fomit-frame-pointer) を行うものの 2 つ, それと -m32 オプション

と -m64 オプションをつけたものそれぞれで行う.

表 1 が最適化無し, 表 2 が速度最適化有りとなる.

CPU/OS	GCC-4.4	GCC-4.6	Micro-C
x86/Linux	7.378	0.829	2.890
x86_64/OS X(-m32)	3.890	0.382	2.288
x86_64/OS X	4.078	0.450	

表 1 アーキテクチャ毎の GCC と micro-c の速度比較（単位: 秒)(最適化無し)

CPU/OS	GCC-4.4	GCC-4.6	Micro-C
x86/Linux	3.252	2.906	2.890
x86_64/OS X(-m32)	1.827	0.934	2.288
x86_64/OS X	1.101	2.896	

表 2 アーキテクチャ毎の GCC と micro-c の速度比較（単位: 秒)(速度最適化)

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 の回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) 与儀健人, 河野真治: “ Continuation based C コンパイラの GCC-4.2 による実装 ”. 琉球大学　情報工学科 学位論文, 2008
4) 与儀健人, 河野真治: “ 組み込み向け言語 Continuation based C の GCC 上の実装 ”. 琉球大学大学院　理工学研究科 学位論文 (修士), 2010
5) 下地篤樹, 河野真治:“ 線形時相論理を用いた Continuation based C プログラムの検証 ”. 琉球大学大学院　理工学研究科 情報工学専攻 学位論文 (修士), 2008
6) 楊挺, 河野真治: “ Continuation based C の実装 ”. 琉球大学大学院　理工学研究科 情報工学専攻 学位論文 (修士), 2002
7) GNU Compiler Collection (GCC) Internals: “ http://gcc.gnu.org/onlinedocs/gccint/ ”