5E-7

汎用クロスコンパイラでの最適化実現法

石川 浩之* 三橋 二彩子**
* 日本電気技術情報システム開発(株)
** 日本電気(株) ﾏｲｸﾛｺﾝﾋﾟｭｰﾀｿﾌﾄｳｪｱ開発本部

1. はじめに

我々は、Retargetableなコンパイラ開発を容易にするコンパイラコード生成ジェネレータCOO(COmpiler Object code generator generator)を実用化し、実際のマイコン向けコンパイラ開発に適用している[1][2]。本稿では、このCOOにより生成されたコンパイラの性能向上に寄与している機種独立最適化（以下最適化と略す）について紹介する。

2. 構成

COOにより生成されたコンパイラのコード生成部の構成を図1に示す。

本最適化は、COOにより自動生成されたコンパイラのコード生成部に組み込まれるライブラリとして存在する。

3. 実現

最適化の対象は、中間木であり、次の最適化項目を実現している。

1) 共通部分式の削除
2) 定数の畳み込み

この最適化を実現するために、最適化の前後処理は次の様に構成されている。

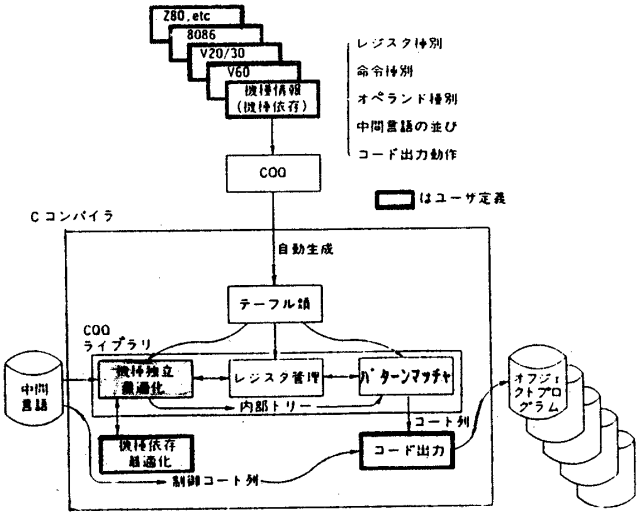


図1 コード生成部の構成

A Portable Machine-independent Optimizer and its Implementation

Hiroyuki ISHIKAWA*, Fusako MITSUHASHI**
* NEC Scientific Information System Development Co. Ltd.
** NEC Corporation Microcomputer Software Development Lab

1. 中間言語の入力
   中間木の構築（基本ブロック単位*）
2. コンマ・後置演算子部分木の分解
   独立な文に分解
3. 評価順序のマーク付け
   関数呼び出し、論理・関係演算子の優先評価
4. 共通部分式の統合
5. 定数の畳み込み
6. パターンマッチャの呼び出し
   コード出力

*) ループ／ラベル等が登場する単位

以下では、4,5,6 について説明する。

① 共通部分式の統合

中間木をボトムアップにたどり、ハッシュテーブルへ登録する（図2－1）。衝突が生じた場合、2つの部分木を照合し、一致したときに一方を解放して1つの部分木に統合し、共通部分式カウンタをインクリメントする（図2－2，2－3，2－4）。この共通部分式には、同形・同値の2種類がある。

共通部分式の認識の範囲は、基本的に複数の文にまたがった中間言語の基本ブロックである。しかし、中間言語中に

- 関数呼び出し
- ポインタ間接の代入

が存在する場合は、この前後で、レジスタの内容が保証されないため、これまでの登録されていたハッシュテーブルを全てクリアする。このような場合、基本ブロック内においても共通性は破壊される。

② 定数の畳み込み

①の同値共通部分式の認識により定数項同士の演算となる場合は、コンパイル時に、この定数演算を畳み込む。例えば、

```
a = 0;
if( a < 10 ) …
```

は、条件式の定数演算が畳み込まれこれに対するコード出力はなされない。

③ コード出力

以上の最適化により、変形された中間木列は、単文毎に切り出されパターンマッチャに渡される。パターンマッチャは、渡された中間木をボトムアップにたどりながら、対応するアセンブラコードを出力し、各葉を、演算結果が格納されたレジスタに置き換えていく。このとき、①で設定した共通部分式カウンタが0でないときは、その葉の参照カウンタをデクリメントし、そのまま残しておく。これにより、①で統合された共通部分式は、演算結果が格納されたレジスタとして、以降参照される。参照カウンタが0のときは、その葉をそのまま削除する（図2－5，2－6）。

以下に共通部分式の削除とコード出力の例を示す（図２参照）。

```
c = a + b;      ..... (1)
d = a + b + c;  ..... (2)
```

上の式において、次の関係が成立する。

同値関係：'a+b' と 'c'

同形関係：(1),(2)式の 'a+b'

共通部分式の削除

図2-1

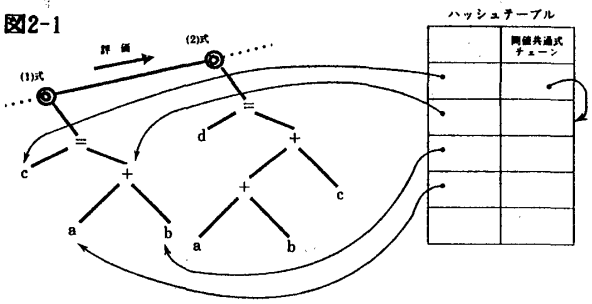


図2-2

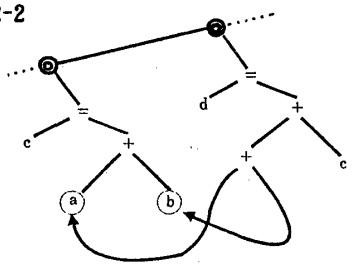


図2-3

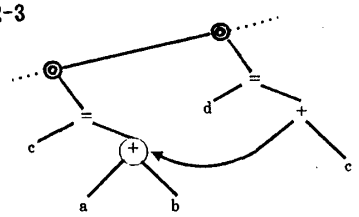
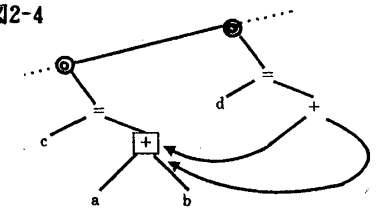


図2-4



コード出力

図2-5

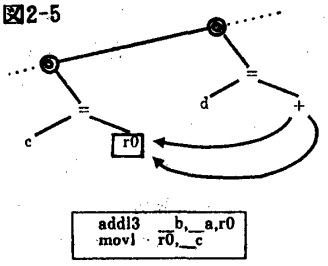


図2-6

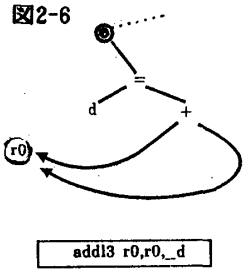


図２　共通部分式の削除とコード出力

表１　性能評価

		V1	V2	V3	V3/V2 (%)	V3/V1 (%)
ackerman.c (23L)	D1	0.6	0.5	0.6	120	100
	D2	188	188	184	97.9	97.9
	D3	3.8	3.9	3.8	97.4	100
arraymerge.c (80L)	D1	2.0	1.8	2.1	117	105
	D2	820	736	688	93.5	83.9
	D3	56.4	52.3	43.3	82.8	76.8
bubblesort.c (60L)	D1	1.6	1.4	1.6	114	100
	D2	520	516	484	93.8	93.1
	D3	32.2	31.4	27.9	88.9	86.6
quick.c (93L)	D1	3.5	2.8	3.4	121	97
	D2	980	968	956	98.8	97.6
	D3	21.4	21.1	20.9	99.1	97.7
sieve.c (31L)	D1	0.9	0.7	0.8	114	89
	D2	352	356	304	85.4	86.4
	D3	2.9	2.9	2.9	100	100

V1:PCC(System-V)　V2:COO版VAX(最適化ﾅｼ)　V3:COO版VAX(最適化ｱﾘ)
D1:ｺﾝﾊﾟｲﾙ(user)時間[ｲﾝﾀﾌﾟﾘﾀ本体](sec)
D2:ｵﾌﾞｼﾞｪｸﾄ(text)ｻｲｽﾞ(byte)　D3:実行時間(sec)
使用ﾏｼﾝ:μVAX-II

4．評価

ＣＯＯを適用して作成したＶＡＸ用Ｃコンパイラを利用して、最適化処理の有無に関してのコンパイル性能をコンパイル時間・オブジェクトサイズ・実行時間の観点から評価を行った。ここでは、ＰＣＣ(Ｕｎｉｘ上のポータブルＣコンパイラ)との相対的な比較も示している。評価プログラムには、５種類のベンチマークプログラムを使用した（表１参照）。

表から次のことがわかる。

① コンパイル時間

共通部分式の削除・定数の畳み込みのために中間木列をたどる回数が増えたためコンパイル時間は、10～20%低下している。

② オブジェクトサイズ・実行時間

改善率は、ソースプログラムに依存するが、今回の改善は、主に共通部分式の削除によるものである。例えば、bubblesort.c では、この削除が10箇所あり、その一部は次のようなものである。

```
cmpl -4(fp),r0          →   cmpl $1,r0

addl3 $1,-16(fp),r1     →   addl3 $1,r0,r1

addl3 $1,-4(fp),r2          pushl r1
pushl r2                →   pushl r0
pushl -4(fp)
```

5．おわりに

今回紹介した最適化は、ＣＯＯを適用したコンパイラに組み込まれ、機種独立な最適化としてコンパイル効率をそれほど落とすことなく、オブジェクト効率の改善が可能である。しかしながら、現在の最適化項目は初歩的なものである。今後は、大域的な最適化を実現し、更に改善率を向上していく予定である。

《参考文献》

[1] 三橋、他　"Code Generator Generator" 情処ﾌﾟﾛｸﾞﾗﾐﾝｸﾞ言語研究会3-2,1985.12
[2] 木村、他　"V60ｱｰｷﾃｸﾁｬを生した言語処理系" 第33回情処全国大会