ハイパフォーマンス
コンピューティング 88－3
（2001．10．26）

ループアンローリングに関するGNU-CのBug Fixと性能改善

佐藤周行†

ループアンローリングは，最適化の手法としては最古の部類に入るものの，loop restructuring時に必要なdataflow analysisなどを必要とせず無条件に適用できることから「易しい最適化」として不当に軽視されてきた．一方，GNU-Cは1980年代半ばからフリーの最適化コンパイラとして高い評価を得てきた．しかし，おおもとの設計思想そのものが古く，モダンな最適化に適応できなくなってきていることを主な原因として，性能面で不十分な結果しか出せない場面が多く見られるようになってきた．Sparc/Solarisにおいてはアンローリングはその典型的な例になっている．

本論文では，Sparc/Solaris上のGNU-Cコンパイラのアンローリングに関するバグを指摘し，そのbug fixをレポートする．修正部分はアンローリング本体に加えて，レジスタ割り付けに深く関係する命令スケジューリングに及んだ．

さらに行列積でその効果を検証した．結果としてアンローリングのバグを直すことでキャッシュミスペナルティの少ない小さいサイズの行列積では46%(大きいサイズでは8.4%)，さらに命令スケジューリングを改善することで50%(大きいサイズでは8.1%)のスピードアップが得られた．

Bug Fix on GNU-C Loop Unrolling, and its Performance Improvement

SATO HIROYUKI†

Loop unrolling, known as one of oldest optimizing techniques, is underestimated because it can anyhow be applied without any dataflow analysis, and therefore it is an "easy" optimization. However, it is not true for modern architectures.

GNU-C has occupied a position of the standard free compiler since its emergence – mid 1980s. However, its performance drawback to leading-edge compilers is now recoverable, which is attributed to the old-fashionedness of RTL, (original) target architectures, and menu of optimizations. Loop unrolling is its typical example.

In this paper, we report bug fixes on loop unrolling of GNU-C compiler on Sparc/Solaris. Bugs are scattered in the part of loop unrolling and instruction scheduling related to register allocation. By fixing those bugs, the performance of matrix multiply has been improved by 46 %(loop unrolling) and further by 50 % (loop unrolling + instruction scheduling).

1. Introduction

ループアンローリングは，最適化の手法としては最古の部類に入るものの，loop restructuring時に必要なdataflow analysisなどを必要とせず無条件に適用できることから「易しい最適化」として不当に軽視されてきた．(古いアーキテクチャでは)性能アップがそれほどのぞめない事実も「軽視」を正当化してきた．しかし，現在主流のCPUアーキテクチャであるSuperScalarやメモリアーキテクチャの主流であるCacheアーキテクチャとの相性の良さが明らかにされ[2]，特にSoftware Pipeliningが本質的に要求することもあってアンローリングは再びその重要性を認識されている．production compilerのほとんどで実効的なループアンローリングが可能になっている．

一方，GNU-Cは1980年代半ばからフリーの最適化コンパイラとして高い評価を得てきた．これは(1980年代半ば当時) Sunの提供するSun3用最適化コンパイラより性能面で大きく上回っていたため，という歴史的な理由による．主流のCPUアーキテクチャがRISC系に変わっても命令スケジューリングに関するルーチンを追加するなどしてベンダ提供のproduction compilerをキャッチアップしようとしてきた．しかし，おおもとの設計思想そのものが古く，モダンな最適化に適応できなくなってきていることを主な原因として，性能面で不十分な結果しか出せない場面が多く見られるようになってきた．

佐藤は授業[1]でGNU-Cの最適化のソースコードを読むことによって最適化コンパイラのstate of the artを講義することを試みたが，コンパイラの内情はモダンな最適化のうち，実装されていないものが複数あり，また古色蒼然としたコードがそのまま残っている部分にあたることも多くあるなど，パフォーマンス上の欠点に対する指摘を裏打ちするものであった．特にアンローリングに関してはGNU-Cの最適化の該当部分は特に不十分である．

しかし，GNU-CがLinuxのdefaultコンパイラとして扱われ，さらにLinuxがRISC系のCPUにも対応してくるようになったことを考えれば，GNU-Cの性能面でのbug fixはある程度の意味がある．

本論文では，Sparc/Solaris上のGNU-Cコンパイラの

† 東京大学 情報基盤センター．
Information Technology Center, The University of Tokyo.

アンローリングに関するバグを指摘し，その bug fix をレポートする．なお，本論文で扱う GNU-C のバージョンは特に断りのない限り 2.96 20000807 とする．

本論文の構成は以下の通りである．3節では GNU-C のアンローリング部分でのバグを指摘し，bugfix とそれに伴う性能向上についてレポートする．4節では，GNU-C でアンローリングの効果を高めるためのさらなる改良について考察する．5節では x86 系に対するアンローリングの効果を検証する．最後に 6節でまとめを与える．

2. Preliminaries

Definition of Loop Unrolling

ここで扱うアンローリングは，GNU-C の定義通り，最内ループのみに適用されるものとする．一部のコミュニティの方言では，以下のものもアンローリングということがある．

```
loop i=1,N                    loop i=1,N,2
  loop j=1,M                    loop j=1,M
    Statement(i,j) ->             Statement(i,j)
 endloop j                        Statement(i+1,j)
endloop i                      endloop j
                              endloop i
```

実は これは (コンパイラの世界での) 標準的な言い方に直せば

[i に関しての loop unrolling] + [j に関しての loop fusion]

である．loop fusion に関して dataflow analysis が必要であり，この意味で最内ループのアンローリングとは異質である．今後はこの形の「アンローリング」は扱わない．

RTL of GCC

GCC で使われている中間言語は RTL とよばれるレジスタ間転送を表現する言語である．論文中で用いる RTL は GCC の出力として得られる S 式表現をそのまま使用した．

3. Unrolling in GCC

GCC において，ループに関係する最適化部分はおそらく最初に書かれたコードのひとつであり，書き方も整理されていない．

特にアンローリングでは，最適化適用範囲の計算に対するバグがあり，正しく動作しない．以下，これを Sparc/Solaris 上の GCC の現象として指摘する．

プログラムとして，簡単のために 以下のものを考える．

```
extern float a[], b[];

loop()
{
        int i;
        for (i = 0; i < 100; i++) {
                a[i] = b[i];
        }
}
```

これを-O3 -funroll-all-loops をつけてコンパイルする．中間表現レベルでのレジスタ割りつけを見る．簡単のためにメモリ移動の部分だけを見る．これから，アンローリングにおいて同一のレジスタが使い回されていることが観察される．

```
...
(insn 106 105 107 (set (reg:SF 115)
;;              レジスタ 115 を使用
        (mem/s:SF (plus:SI (reg:SI 121)
                (reg:SI 111)) 5)) -1 (nil)
    (nil))

(insn 107 106 109 (set (mem/s:SF
                           (plus:SI (reg:SI 121)
                (reg:SI 107)) 5)
        (reg:SF 115)) -1 (nil)
    (nil))
...
(insn 114 113 115 (set (reg:SF 115)
;;              レジスタ 115 を使い回し
        (mem/s:SF (plus:SI (reg:SI 125)
                (reg:SI 111)) 5)) -1 (nil)
    (nil))

(insn 115 114 117 (set (mem/s:SF
                           (plus:SI (reg:SI 125)
                (reg:SI 107)) 5)
        (reg:SF 115)) -1 (nil)
    (nil))
...
(insn 122 121 123 (set (reg:SF 115)
;;              レジスタ 115 をまた (!) 使い回し
        (mem/s:SF (plus:SI (reg:SI 127)
                (reg:SI 111)) 5)) -1 (nil)
    (nil))

(insn 123 122 125 (set (mem/s:SF
                           (plus:SI (reg:SI 127)
                (reg:SI 107)) 5)
        (reg:SF 115)) -1 (nil)
    (nil))
...
```

結果としての出力コードもこれを反映している．配列のインデックスに使われるレジスタと違い，ロード / ストア

に使うレジスタに %f2 のみが使われていることが観察できる．

```
        ld      [%o2+%o3], %f2
        add     %o2, 4, %o0
        st      %f2, [%o2+%o4]
        ld      [%o0+%o3], %f2
###           レジスタ %f2 を使用
        add     %o2, 8, %o1
        st      %f2, [%o0+%o4]
        ld      [%o1+%o3], %f2
###           レジスタ %f2 を使い回し
        add     %o2, 12, %o0
        st      %f2, [%o1+%o4]
        ld      [%o0+%o3], %f2
###           レジスタ %f2 をまた (!) 使い回し
        add     %o2, 16, %o1
        st      %f2, [%o0+%o4]
...
```

この現象は吉田によって授業[1]に関係した場面で報告され[3]，修士論文[4]でも，実験環境報告の一部として触れられた．

筆者はしかしこれは Sparc/Solaris に限って観察される現象であることを確認し，このバグを修正した．

バグは，Sparc ではループアンローリングによってコード量が展開段数倍以上に増加する可能性があることを見逃してアンロールされたループの範囲の上下限が正しく認識されない所に存在した．その結果としてループ内部での繰り返しをまたがる変数かどうかののチェックがなされない場合がある．繰り返しをまたがらないという保証がなければレジスタ割当の自由度が減り，アンロールの効果があがらない．

bug fix の結果，中間言語レベルで以下のコードが出るようになった．レジスタが繰り返しに局所的かどうかのチェックが正しくされるようになり，ロード / ストアに使うレジスタがこのレベルでは分散していることが観察できる．

```
Loop unrolling: 100 iterations.
Unrolling loop 10 times.
...
(insn 106 105 107 (set (reg:SF 125)
            レジスタ 125 使用
        (mem/s:SF (plus:SI (reg:SI 121)
                (reg:SI 111)) 5)) -1 (nil)
    (nil))

(insn 107 106 109 (set (mem/s:SF (plus:SI (reg:SI 121)
                (reg:SI 107)) 5)
        (reg:SF 125)) -1 (nil)
    (nil))
...
(insn 114 113 115 (set (reg:SF 128)
        レジスタ 125 以外のレジスタ 使用
        (mem/s:SF (plus:SI (reg:SI 126)
                (reg:SI 111)) 5)) -1 (nil)
    (nil))

(insn 115 114 117 (set (mem/s:SF (plus:SI (reg:SI 126)
                (reg:SI 107)) 5)
        (reg:SF 128)) -1 (nil)
    (nil))
...
(insn 122 121 123 (set (reg:SF 131)
        レジスタ 125, 128 以外のレジスタ使用
        (mem/s:SF (plus:SI (reg:SI 129)
                (reg:SI 111)) 5)) -1 (nil)
    (nil))

(insn 123 122 125 (set (mem/s:SF (plus:SI (reg:SI 129)
                (reg:SI 107)) 5)
        (reg:SF 131)) -1 (nil)
    (nil))
```

しかし，GCC ではレジスタ割り付けがかなり貧弱であり，結局出てくるコードは 2 段程度のアンローリングの効果しか持たないことが観察される．GNU-C の基本ブロックを対象にしたレジスタ割り付けでは，生存区間がバッティングしないものには，同一のレジスタを使い回す可能性が高いアルゴリズムを使っている．

実際，ロードの対象になるレジスタに対する割り付け方は以下のようになっている (中間表現から抜粋)．各数字はレジスタ番号を表している．

Pseudo Register#	125	128	131	134	137
⇓(Allocate)					
Real Register#	34	35	34	35	34

	140	143	146	149	152
⇓(Allocate)					
	35	34	36	35	34

出てきたコードはこれを反映せざるをえない．つまり:

```
        ld      [%i5+%g1], %f2
        add     %i5, 4, %i0
        st      %f2, [%i5+%o7]
        add     %i5, 8, %i1
        ld      [%i0+%g1], %f3
                    (%f2 と異なるレジスタ使用)
        add     %i5, 12, %i2
        ld      [%i1+%g1], %f2
```

```
                    (%f2 に回帰)
st      %f3, [%i0+%o7]
st      %f2, [%i1+%o7]
add     %i5, 16, %i3
ld      [%i2+%g1], %f3
                (ひとつおいて %f3 に回帰)
add     %i5, 20, %i0
ld      [%i3+%g1], %f2
                (ひとつおいて %f2 に回帰)
st      %f3, [%i2+%o7]
st      %f2, [%i3+%o7]
add     %i5, 24, %i4
ld      [%i0+%g1], %f3
add     %i5, 28, %i2
ld      [%i4+%g1], %f2
st      %f3, [%i0+%o7]
add     %i5, 32, %i1
ld      [%i2+%g1], %f4
                RealRegister 36 に対応
st      %f2, [%i4+%o7]
add     %i5, 36, %i0
ld      [%i1+%g1], %f3
st      %f4, [%i2+%o7]
ld      [%i0+%g1], %f2
st      %f3, [%i1+%o7]
st      %f2, [%i0+%o7]
```

それでもパフォーマンスは修正前と比べて上がり，多少ではあるがアンローリングの効果を実感できるようになった．これを行列積 (ループは ikj) を使ったベンチマークで示す．配列のサイズは 256 と 1000 について測定した．manual unroll の段数 は GNU-C のアンローリングの段数と同じにして計測した．すなわち，サイズ 256 では 8 段，1000 では 5 段である．

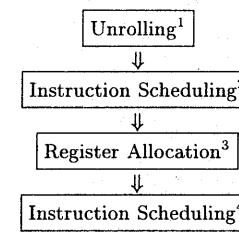
	256	fixed	1000	fixed
unroll なし	52.4	52.4	30.7	30.7
unroll あり	55.9	81.8	31.8	34.5
manual unroll	–	72.9	–	37.5

(Sun E3500[Ultra-Sparc II(336MHz)] in MFLOPS)

4. Further Improvement

アンローリングは上手に使えば性能向上に非常に有効である．そのために，段数を決定する時にはレジスタ資源を考えることが必要である．すなわち，アンロールすることによるレジスタプレッシャの増大をレジスタスピルコードが生じる手前までで押えるために，レジスタ割り付けのフェーズと interaction を持たなければならない．レジスタ割り付けは命令スケジューラと本質的に関係するから，結局この三者は適当に収束するまで繰り返し行なうか，またはこれらを同時に扱う最適化ルーチンを新たに書き起こさなければならない．

現在 GNU-C の最適化部分の構成では，それぞれが独立に書かれ，以下に示す順に適用されることになっている．



この構成は GNU-C にとって致命的な欠陥になっている．実際，アンローリングに関する bugfix を行なっても，コードが optimal にならないのは 2.96 においては次が観察されるからである．

(1) 最初に呼ばれるスケジューラは，実行優先度を次のように設定している．
  (a) その命令に依存する後続命令の長さの長いもの
  (b) レジスタプレッシャの小さいもの
  (c) 依存関係の多いもの．
  この部分でレジスタプレッシャの小さいものを優先するために，store 系の命令が優先してスケジュールされる☆．

(2) 次のレジスタ割り付けでは，レジスタプレッシャを小さくするように割り付ける．この時点でレジスタの使い回しが発生する．

(3) 最後のスケジューリングにおいては，命令間に (レジスタ割り付けで発生した) false dependence が存在するので，もはや調節ができない．

ここで，レジスタ割り付け部分とスケジューラは適切に情報をやりとりしてレジスタ・メモリ間の bandwidth をできるだけ使いきり，なおかつ spill code が発生しないようにすべきであった．

そこで，次善の策として，最初の scheduling のフェーズ (上図中 2) において使用レジスタの数が少ない時は，レジスタプレッシャを増やす命令 (具体的に load 命令) の優先度を上げるようにスケジューラに変更を加えた．具体的には以下のとおりである．

```
if (#Live レジスタ < # レジスタ /2) {
    switch (命令) {
    case load:
        命令の優先度 += 128;
```

☆ 2.95 では，さらにこれを調整して，
- レジスタ の生存区間ができるだけ長くなるもの

を優遇している．このために store 系の命令が後ろにまわり，不要な spill code が頻発した．
2.96 では，この調整を外している．不要な spill code が発生しなくなった代わりにレジスタ・メモリ間の bandwidth を使い切らないようなレジスタの使い回しが発生した．

```
    case store:
        命令の優先度 /= 4;
    }
} else if (#Live レジスタ < # レジスタ) {
    switch (命令) {
    case load:
        命令の優先度 += 8;
    case store:
        命令の優先度 /= 2;
    }
else /* Register Pressure 大 */
        /* Do Nothing */
}
```

この変更が2.95と異なるのはstoreの優先度を下げることよりもloadの優先度を上げていることを重要視する点である．loadの優先度を上げても結果的にレジスタのlifeが長くなる点で2.95の戦略と重なるが，storeを後ろに下げすぎないため，spill codeの発生確率は2.95と比較して低い．

改善の結果，次のコードが生成されるようになった．

```
...
add %o1, 32, %i1
add %o1, 36, %i0
ld [%o1+%o2], %f2
addcc %o4, -10, %o4
ld [%o0+%o2], %f11
ld [%o7+%o2], %f10
ld [%g1+%o2], %f9
ld [%i5+%o2], %f8
ld [%i4+%o2], %f7
ld [%i3+%o2], %f6
ld [%i2+%o2], %f5
ld [%i1+%o2], %f4
ld [%i0+%o2], %f3
st %f2, [%o1+%o3]
st %f11, [%o0+%o3]
st %f10, [%o7+%o3]
st %f9, [%g1+%o3]
st %f8, [%i5+%o3]
st %f7, [%i4+%o3]
st %f6, [%i3+%o3]
st %f5, [%i2+%o3]
st %f4, [%i1+%o3]
add %o1, 40, %o1
bpos .LL6
st %f3, [%i0+%o3]
...
```

10段アンロールされ，レジスタ割り付けもきれいにばらまかれていることが観察できる．

行列積(条件は3節のものと同一)で性能向上を見る．

サイズ	256	fixed	1000	fixed	*
unroll なし	52.4	52.4	30.7	30.7	1
unroll あり	55.9	81.8	31.8	34.5	2
scheduler 改善	–	83.9	–	34.4	4

(MFLOPS)

* 実質的なアンローリング段数

若干の性能向上が見られるコードがあることが分かった．ここでいう「実質的なアンローリング段数」とは，レジスタの使い回しの点から見て実際にアンロールされているのと同じ効果を持つ段数のことである．

5. Unrolling in x86 Architectures

このbug fixはRISC系のものに対してのみ有効である．x86系では，レジスタ割り付けが(中間言語レベルで)最適であっても，その後のコード生成部分で吸収されてしまい，結局(GCCの中では)効果がない．

ここではGNU-Cの出すx86をターゲットとしたコードを検証する．使用プログラムは同一．

中間言語レベルでは，レジスタの分散はうまく行っている．

```
(insn 46 16 47 (parallel[
            (set (mem/s:SF (plus:SI (mult:SI (reg/v:SI 21)
                                (const_int 4))
                            (symbol_ref:SI ("a"))))
                 (mem/s:SF (plus:SI (mult:SI (reg/v:SI 21)
                                (const_int 4))
                            (symbol_ref:SI ("b")))))
            (clobber (scratch:SI))
        ] ) -1 (nil)
    (nil))
...
(insn 49 48 50 (parallel[
            (set (mem/s:SF (plus:SI (mult:SI (reg:SI 25)
                                (const_int 4))
                            (symbol_ref:SI ("a"))))
                 (mem/s:SF (plus:SI (mult:SI (reg:SI 25)
                                (const_int 4))
                            (symbol_ref:SI ("b")))))
            (clobber (scratch:SI))
        ] ) -1 (nil)
    (nil))
...
```

しかし，実レジスタに割り付ける段になると，x86のレジスタ数の本質的な少なさから，平凡なコードになる．

```
...
movl %eax,a(,%edx,4)
```

```
movl b+4(,%edx,4),%eax
movl %eax,a+4(,%edx,4)
movl b+8(,%edx,4),%eax
movl %eax,a+8(,%edx,4)
movl b+12(,%edx,4),%eax
movl %eax,a+12(,%edx,4)
movl b+16(,%edx,4),%eax
...
```

さらに，浮動小数点数演算として x87 系の演算を指定すると，GNU-C のロジックを使う限り，レジスタスタックの性質から同じレジスタを使い回さざるを得ず，アンローリングの効果がまったく期待できない．下の結果は Pentium 266MHz で行列積を測定したものである．GCC のバージョンは 2.7.2．コンパイルオプション `-funroll-all-loops` では，配列サイズ 256 に対して 4 段に unroll される．このオプションのありなし，manual unroll についてそれぞれ性能を測定した．manual unroll は manual で 4 段に展開したものについて測定したものである．

unrolling なし	17.2MFLOPS
unrolling あり	17.9MFLOPS
manual unroll	15.8MFLOPS

ただし，他のコンパイラではレジスタスタックの使い回しを工夫してそれなりの性能向上が見られる．下は富士通の C コンパイラ (Version 1.0 May 19 1999) での測定．`-Kfast` で 8 段の unroll を行っている．manual unroll もそれに併せて 8 段の unroll を行っている．

`-Kfast` なし	25.8MFLOPS
`-Kfast` あり	30.2MFLOPS
manual unroll	29.2MFLOPS

6. Concluding Remarks

本論文ではまず GNU-C のアンローリング部分でのバグを指摘し，bugfix とそれに伴う性能向上についてレポートした．次に，bugfix とは独立に 5節では x86 系に対するアンローリングの効果を検証し，効果が薄いことを実証した．

また，4節では GNU-C でアンローリングの効果を高めるためのさらなる改良について考察した．ここではアンローリング部分の bug がなくなると，アンローリングにおいて多量のレジスタを消費することから今度はレジスタ割り付けの性能の悪さ，および，両者の interaction のなさからくるアンローリング段数の不適切な決定が新たに問題になることを指摘した．そこでスケジューラに改善を加え，レジスタ・メモリ間の bandwidth を広く保つようにスケジューラにヒントを与えるようにした．なお，これら修正部分は筆者の Web ページ[5] から download 可能である．

また，x86 系に対してアンローリングの効果が期待できないことは unrolling への関心が一部の人間に限定されることを意味する．x86 系向けへの unrolling の改善は今後の課題である．

参 考 文 献

1) 佐藤周行: コンパイラ構成法特論 I, 東京大学 大学院 (理学系) 講義科目, 2000.
2) Sato, H., Yoshida, T.: Characteristics of Loop Unrolling Effect: Software Pipelining and Memory Latency Hiding, Proc. Int'l Workshop on Innovative Architecture 2001, to appear.
3) 吉田映彦: Personal Communication, 2000.
4) Yoshida, T.: Characteristic Extraction of Loop Unrolling and its Modeling, Univ. Tokyo Master Thesis for Master of Science, 2001.
5) `http://www.super-computing.org/~schuko/gcc.html`