ハイパフォーマンス
コンピューティング
（２００２．５．２７）
90－5

# PCアーキテクチャ向け命令スケジューリングにおけるデータスタックの仮想多重使用とその効果

長倉裕叔[†]　梅谷征雄[††]

PC アーキテクチャの代表的なプロセッサの１つに PentiumⅢが挙げられる．しかしながら，Pentium のアセンブリ命令は FPU（Floating Point Unit）データスタックの依存によって FPU 命令のスケジュールが困難である．そこで，データスタックを仮想多重使用することを考案し，遺伝的命令スケジューラやリストスケジューラを使用した実験を行なった．その結果，Borland C++ Builder4 コンパイラと比べ，最大 26.5 %の性能向上を確認した．

# Virtual Multiplex Use of the Data Stack for PC Architecture Oriented Instruction Scheduling and its Effect

Hirotoshi Nagakura[†] and Yukio Umetani[††]

PentiumⅢ is one of the typical processors of PC architecture. But the instruction scheduling of FPU instructions is constrained by the existence of FPU register stack. To relax its constraint, we devised the virtual multiplex use of FPU stack. As a result, the performance evaluted using 24 Livermore kernels on PentiumⅢ machines was improved up to 26.5% compared to the conventional scheduling.

## 1. はじめに

PC の代表的なプロセッサとして Pentium がある．Pentium アーキテクチャでは FPU（Floating Point Unit）データスタック[1)] の存在が，FPU 命令の命令スケジューリングのネックとなっており，それは式の長い演算やループアンロールコードでは顕著である．例えば，図 1 は浮動小数点演算を行なうループを 2 回展開した UNROLL コードの一部である．また，図 2 は，図 1 のアセンブリ命令列である．図 2 において，fld は浮動小数点のロード命令であり，FPU データスタックのトップにデータを PUSH する FPU 命令である．それから，fmul や fadd は浮動小数点の算術演算を行なう FPU 命令であり，スタックのトップとオペランドで指定されたデータとの計算を行ない，計算結果をスタックのトップに書き込む．そして，fstp は浮動小数点のストア命令であり，FPU データスタックのトップにデータをストアした後，POP する FPU 命令である．このように，Pentium の FPU 命令は FPU データスタックのトップと依存を持つことが多い．そのため，図 2 の依存グラフは図 3(a) のように，移動可能な命令が少なくなりがちである．図 1 はループ展開によって計算が並列となっているが，スタックによる依存のために FPU 命令が入れ替えられない．これでは，ループ展開を行なっても並列度が上がらず，ループを展開することによって得られる効果は非常に小さいと予想される．

Pentium アーキテクチャに関してはこれまでロード・ストア命令の Out of Order 実行が有効であることは知られている[1)] が，実数演算に関してはデータスタックによる拘束がスケジューリングの障害となる．そこで，筆者らは２章で述べるようにデータスタックを仮想多重使用することで，上記の障害を克服することを思いついた．

そして，仮想多重使用の機能を Pentium のスタック入れ替え命令を使って実現した．それによって，図 2 の依存グラフは図 3(b) となる．すなわち，スタックによる依存が無くなり，FPU 命令のスケジュールが可能となっている．またスタック入れ替え命令の自動挿入アルゴリズムを考案し，アセンブラのコンバータ

† 静岡大学大学院　情報学研究科　情報学専攻
Department of Information, Graduate School of Information, Shizuoka University
†† 静岡大学　情報学部　情報科学科
Department of Information Science, Faculty of Information, Shizuoka University

```
double da, dx[ ], dy[ ];
int i, m, n;

for (i = m; i <n; i = i + 2)
{dy[i] = dy[i] + da*dx[i];
dy[i+1] = dy[i+1] + da*dx[i+1];
}
```

**図 1** プログラムの例
Fig. 1 An example of a program.

```
1   @1:
2           fld     qword ptr [ebp+12]
3           fmul    qword ptr [ecx]
4           add     edx, 2
5           fadd    qword ptr [eax]
6           fstp    qword ptr [eax]
7           fld     qword ptr [ebp+12]
8           fmul    qword ptr [ecx+8]
9           add     ecx, 16
10          fadd    qword ptr [eax+8]
11          fstp    qword ptr [eax+8]
12          add     eax, 16
13          cmp     esi, edx
14          jg      short @1
```

**図 2** 図 1 のアセンブリ命令列
Fig. 2 Assembly instruction sequence of Figure 1.

として実装した．これを遺伝的命令スケジューラ[2)~4)]とリストスケジューラ[5),6)]に組み込み，LFK[7)]を対象にそれらを評価し，その有効性を確認した．

## 2. データスタックの仮想多重使用による FPU データスタック依存の改善

ここでは，データスタックの仮想多重使用について述べる．

### 2.1 仮想多重使用の実現方式

まず，SDN (Stack Data Number) というパラメータを各命令に設定する．SDN の値は，命令が生成または参照するスタック上のデータの有効範囲に番号付けをしたものである．図 2 の命令列では，命令 2・3・5・6，命令 7・8・10・11 のおのおのに独立したデータ有効範囲がある．この場合，命令 2・3・5・6 の SDN 値を 1，同様に命令 7・8・10・11 の SDN を 2 であると設定する．すなわち，SDN 値が 1 以上

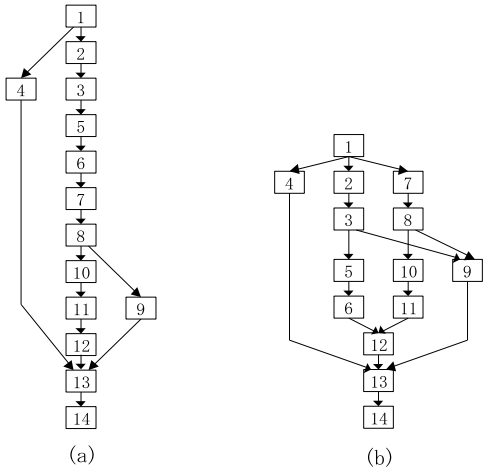


**図 3** 図 2 の依存グラフ
Fig. 3 Dependence graph of Figure 2.

であり，同じ値を持つ命令間にはデータによる依存があるとする．また，スタック上のデータを 2 つ必要とする命令ならば，その命令は 2 つの SDN を持つ．

次に，SDN の割り付け方法について述べる．FPU 命令は SDN の値が 1 以上，それ以外の命令は SDN が 0 と設定する．SDN による依存生成は SDN が 1 以上の命令に対して行なう．SDN を割り付けるために，あらかじめ作業用スタックと変数 $i$ を用意する．$i$ の初期値は 0 とする．そして，以下の手順で SDN を設定していく．

**手順 1：** 命令が FPU 命令でない，つまりデータスタックを使用しない命令であれば，SDN を 0 とする．

**手順 2：** FPU 命令であり，データスタックにデータを PUSH する命令であれば，$i$ の値を 1 増やす．そして，その $i$ の値を作業用スタックに PUSH し，SDN を $i$ とする．

**手順 3：** FPU 命令であり，データスタックにデータを PUSH する命令でなければ，作業用スタックのトップにある値を SDN とする．その命令がスタック上のデータを 2 つ必要とする命令ならば，必要とするデータが作業用スタックのどの位置に存在しているのかを調べ，その位置に格納されている値を SDN に設定する．

**手順 4：** FPU 命令であり，データスタックのデータを POP する命令であれば，作業用スタックを POP する．

**手順 5：** 対象とする命令列が終わってなければ，手順 1 へ戻り，次の命令の SDN を設定する．

そして，スタックによる依存を削除する代わりに，

```
5       fadd    qword ptr [eax]
7       fld     qword ptr [ebp+12]
6       fstp    qword ptr [eax]

                挿入前

5       fadd    qword ptr [eax]
7       fld     qword ptr [ebp+12]
        fxch    st(1)
6       fstp    qword ptr [eax]

                挿入後
```

**図 4** fxch 挿入例
Fig. 4 An example of fxch insertion.

このSDNを使って依存を生成することとした．

例えば，図2ならば，スタックによる依存を使用して依存グラフを作成すると，依存グラフは図3(a)となる．一方，スタック依存の代わりにSDNの値が1以上のデータ依存を使用すると図3(b)となるのである．

しかしながら，その依存グラフを用いてスケジューリングすると，FPUデータスタックがプログラムで意図するように使用されていないコードを生成することがある．それは，スタックによる依存を仮想的に削除した依存グラフを使用したため，正しくないデータを参照する恐れがあるからである．そこで，スタック入れ替え命令fxch命令[8]を挿入し，次の命令が必要としているデータをスタック上の正しい位置に移動させることにした．その命令はスタックの内容を入れ替える命令である．例えば，図4のように，fxch命令を挿入する．挿入前は命令7でロードしたデータを命令6でストアするという間違った動作になっている．だが，fxch命令の挿入によって，スタックデータが入れ替わり，命令5の結果を命令6がストアするという正しい動作にすることができる．そのように，間違った動作になる箇所を見つけ出し，fxch命令を挿入し，正しい動作に修正するアルゴリズムを作成した．fxch命令の自動挿入法に関しては2.3で詳しく述べる．

### 2.2 SDNに基づく依存グラフの生成とスケジューリング

SDNの値が1以上である命令は同じ値を持つ命令同士で依存がある．その依存と他の依存とを合わせ，依存グラフを作成する．ただし，FPUデータレジスタは8個であるため，データスタックは8つまでしかデータを積めない．しかしながら，仮想多重使用によって，並列度が増すため，9個以上積んでしまう命令列になる可能性がある．それを防ぐため，データをPUSHする命令とPOPする命令をチェックしておき，$a(a=1,2,3,\cdots,n)$番目にPOPする命令と（$a+8$）番目のPUSHする命令との間に依存を作成した．それにより，9個以上データを積んでしまう状況を防ぐことができる．これらの依存関係を基に作成した依存グラフの制約内でスケジューリングを行なう．

### 2.3 fxchの自動挿入法

スケジューリング後は作業用スタックを用意し，以下の手順でfxch命令を挿入する．これにより，それぞれのFPU命令が正しいデータを扱えるように調整するのである．

**手順1：** 命令がFPU命令であり，データスタックにデータをPUSHするものであれば，作業用スタックにそのSDNをPUSHする．

**手順2：** 命令がFPU命令であり，さらにデータスタックにデータをPUSHする命令でなければ，その命令のSDNと作業用スタックのトップのSDNの値を比較する．それらの値が等しければ，fxch命令を挿入する必要はない．しかし，異なる場合は，作業用スタックのトップにそのSDNの値がくるようにfxch命令を挿入する．SDNを2つ持つ命令ならば，残りのSDNで指定されているデータを正しく扱えるように，fxch命令の挿入又は命令のオペランドの書き換えを行なう．その後，挿入したfxch命令通りに作業用スタックの値を交換する．

**手順3：** FPUデータスタックをPOPする命令である場合は，作業用スタックをPOPする．

**手順4：** 対象とする命令列が終わってなければ，次の命令へ行き，手順1へ戻る．

## 3. スケジューラ

本研究で使用する遺伝的命令スケジューラとリストスケジューラに関して，説明する．

### 3.1 遺伝的命令スケジューラ

遺伝的命令スケジューラは，遺伝的アルゴリズム[9),10)]を用いた命令スケジューリングである．静的な評価関数を用いる従来のコンパイル手法とは異なり，対象としたプログラムの実行時間のみに依存したスケジューリングをするため，そのプロセッサの最適化方法が分からなくとも最適化できる．今回は，Pentiumのアセンブリ命令コードに適用した遺伝的命令スケジューラを作成した．

**表 1** 実験結果
Table 1 Performance table.

| LFK No. | BCC (ms) | GA (ms) | GA-ST (ms) | list (ms) | list-ST (ms) | 改良度 GA (%) | 改良度 GA-ST (%) | 改良度 list (%) | 改良度 list-ST (%) | IS_ Num |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1675 | 1655 | 1656 | 1665 | 1700 | 1.2 | 1.1 | 0.6 | -1.5 | 15 |
| 2 | 733 | 704 | 705 | 737 | 805 | 4.0 | 3.8 | -0.5 | -9.8 | 38 |
| 3 | 3049 | 3050 | 3049 | 3174 | 3174 | 0.0 | 0.0 | -4.1 | -4.1 | 20 |
| 4 | 441 | 434 | 433 | 444 | 444 | 1.6 | 1.8 | -0.7 | -0.7 | 25 |
| 5 | 605 | 596 | 601 | 609 | 609 | 1.5 | 0.7 | -0.7 | -0.7 | 19 |
| 6 | 224 | 222 | 221 | 224 | 224 | 0.9 | 1.3 | 0.0 | 0.0 | 25 |
| 7 | 959 | 949 | 844 | 996 | 894 | 1.0 | 12.0 | -3.9 | 6.8 | 28 |
| 8 | 194 | 189 | 182 | 192 | 185 | 2.6 | 6.2 | 1.0 | 4.6 | 73 |
| 9 | 1554 | 1533 | 1395 | 1555 | 1443 | 1.4 | 10.2 | -0.1 | 7.1 | 31 |
| 10 | 2685 | 2295 | 2292 | 2343 | 2343 | 14.5 | 14.6 | 12.7 | 12.7 | 73 |
| 11 | 301 | 301 | 301 | 307 | 307 | 0.0 | 0.0 | -2.0 | -2.0 | 20 |
| 12 | 315 | 314 | 314 | 315 | 314 | 0.3 | 0.3 | 0.0 | 0.3 | 9 |
| 13 | 278 | 276 | 271 | 279 | 274 | 0.7 | 2.5 | -0.4 | 1.4 | 74 |
| 14 | 644 | 623 | 617 | 639 | 658 | 3.3 | 4.2 | 0.8 | -2.2 | 122 |
| 15 | 804 | 803 | 801 | 805 | 804 | 0.1 | 0.4 | -0.1 | 0.0 | 226 |
| 16 | 344 | 344 | 309 | 345 | 343 | 0.0 | 10.2 | -0.3 | 0.3 | 100 |
| 17 | 1124 | 1108 | 1104 | 1107 | 1239 | 1.4 | 1.8 | 1.5 | -10.2 | 80 |
| 18 | 254 | 252 | 269 | 254 | 293 | 0.8 | -5.9 | 0.0 | -15.4 | 223 |
| 19 | 811 | 804 | 805 | 804 | 805 | 0.9 | 0.7 | 0.9 | 0.7 | 39 |
| 20 | 913 | 877 | 875 | 913 | 920 | 3.9 | 4.2 | 0.0 | -0.8 | 86 |
| 21 | 854 | 841 | 841 | 874 | 874 | 1.5 | 1.5 | -2.3 | -2.3 | 10 |
| 22 | 824 | 824 | 822 | 824 | 824 | 0.0 | 0.2 | 0.0 | 0.0 | 41 |
| 23 | 576 | 559 | 559 | 599 | 585 | 3.0 | 3.0 | -4.0 | -1.6 | 69 |
| 24 | 549 | 549 | 548 | 549 | 549 | 0.0 | 0.2 | 0.0 | 0.0 | 19 |
| 平均 | | | | | | 1.9 | 3.1 | -0.1 | -0.7 | |

### 3.2 リストスケジューラ

このリストスケジューラは各命令毎に設定された進行優先度を用いてスケジュールする．進行優先度は，その命令が依存グラフにおいて，どの位置にあるのかを示すものである．その値は始めの方で実行されるべき命令は値が小さく，最後の方で実行されるべき命令は値が大きい．このスケジューリングの目的は，早めに実行しておく命令を判断し，スケジュールすることである．そして，進行優先度の小さい命令を優先的にスケジュールしていく．

## 4. 有効性評価

ここでは遺伝的命令スケジューラとリストスケジューラに対し，仮想多重使用とスタック入れ替え命令の自動挿入を組みこみ，LFK を利用して評価を行なう．遺伝的命令スケジューラは，個体数 10，世代数 30 で使用した．LFK は FORTRAN で書かれた 24 個のカーネルループプログラムから構成されている．今回の実験では，FORTRAN から C に書き直したものを使用した．実験で使用したマシンは GATEWAY Performance800（メモリ 192MB）であり，PentiumⅢ800MHz を搭載している．OS は Window98SE である．コンパイラは Borland C++ Builder4 を使用し，そのコンパイラで最適化オプションを用いて作成したアセンブリ命令列を対象にスケジュールした．また，実行時間の評価には Borland C++ Builder4 にある clock 関数を使用した．clock 関数は２つのイベント間の時間を ms 単位で計測できるものである．また，各スケジューラには，依存グラフ作成時にフラグレジスタやメモリによる不必要な依存関係を可能な範囲で取り除く機能を施してある[4),11)]．

### 4.1 評価結果

表 1 は kernel1〜24 の実験結果であり，表 2 は kernel1〜24 の C ソースコードにおいてループ間のデータ依存の少ない kernel を対象に実験した結果である．NOROLL はループ展開をしていないものであり，UNROLL は C ソースレベルでループ展開したものである．UNROLL の後に続く 2〜4 の数値はそれぞれ何回のループ展開をしたかを示している．測定時の誤差を小さくするため，実験結果の実測値は 10 回実行した平均値である．表 1 と表 2 において，

LFKNo. :kernel の番号

BCC:Borland C++ Builder4 によるコンパイラ及び最適化 (スケジュール前のコード)

**表 2** 実験結果 (UNROLL)
Table 2 Performance table(UNROLL).

| LFK No. | BCC (ms) | GA (ms) | GA-ST (ms) | list (ms) | list-ST (ms) | 改良度 GA (%) | 改良度 GA-ST (%) | 改良度 list (%) | 改良度 list-ST (%) | IS_ Num |
|---|---|---|---|---|---|---|---|---|---|---|
| k1_NOROLL | 1675 | 1655 | 1656 | 1665 | 1700 | 1.2 | 1.1 | 0.6 | -1.5 | 15 |
| k1_UNROLL2 | 1615 | 1596 | 1539 | 1624 | 1604 | 1.2 | 4.7 | -0.6 | 0.7 | 23 |
| k1_UNROLL3 | 1638 | 1594 | 1589 | 1670 | 1685 | 2.7 | 3.0 | -2.0 | -2.9 | 31 |
| k1_UNROLL4 | 1650 | 1569 | 1539 | 1581 | 1554 | 4.9 | 6.7 | 4.2 | 5.8 | 39 |
| k7_NOROLL | 959 | 949 | 844 | 996 | 894 | 1.0 | 12.0 | -3.9 | 6.8 | 28 |
| k7_UNROLL2 | 954 | 948 | 701 | 957 | 734 | 0.6 | 26.5 | -0.3 | 23.1 | 48 |
| k7_UNROLL3 | 952 | 946 | 711 | 954 | 769 | 0.6 | 25.3 | -0.2 | 19.2 | 68 |
| k7_UNROLL4 | 949 | 941 | 700 | 955 | 719 | 0.8 | 26.2 | -0.6 | 24.2 | 88 |
| k8_NOROLL | 194 | 189 | 182 | 192 | 185 | 2.6 | 6.2 | 1.0 | 4.6 | 73 |
| k8_UNROLL2 | 176 | 173 | 166 | 174 | 170 | 1.7 | 5.7 | 1.1 | 3.4 | 133 |
| k8_UNROLL3 | 170 | 170 | 159 | 174 | 164 | 0.0 | 6.5 | -2.4 | 3.5 | 193 |
| k8_UNROLL4 | 170 | 166 | 158 | 172 | 160 | 2.4 | 7.1 | -1.2 | 5.9 | 253 |
| k9_NOROLL | 1554 | 1533 | 1395 | 1555 | 1443 | 1.4 | 10.2 | -0.1 | 7.1 | 31 |
| k9_UNROLL2 | 1532 | 1504 | 1369 | 1505 | 1405 | 1.8 | 10.6 | 1.8 | 8.3 | 57 |
| k9_UNROLL3 | 1508 | 1494 | 1374 | 1501 | 1395 | 0.9 | 8.9 | 0.5 | 7.5 | 83 |
| k9_UNROLL4 | 1494 | 1489 | 1354 | 1513 | 1396 | 0.3 | 9.4 | -1.3 | 6.6 | 109 |
| k12_NOROLL | 315 | 314 | 314 | 315 | 314 | 0.3 | 0.3 | 0.0 | 0.3 | 9 |
| k12_UNROLL2 | 251 | 251 | 189 | 253 | 220 | 0.0 | 24.7 | -0.8 | 12.4 | 12 |
| k12_UNROLL3 | 209 | 167 | 167 | 210 | 230 | 20.1 | 20.1 | -0.5 | -10.0 | 15 |
| k12_UNROLL4 | 189 | 173 | 157 | 189 | 204 | 8.5 | 16.9 | 0.0 | -7.9 | 18 |
| k18_NOROLL | 254 | 252 | 269 | 254 | 293 | 0.8 | -5.9 | 0.0 | -15.4 | 223 |
| k18_UNROLL2 | 304 | 294 | 275 | 307 | 290 | 3.3 | 9.5 | -1.0 | 4.6 | 294 |
| k18_UNROLL3 | 304 | 296 | 265 | 307 | 290 | 2.6 | 12.8 | -1.0 | 4.6 | 362 |
| k18_UNROLL4 | 304 | 299 | 269 | 305 | 271 | 1.6 | 11.5 | -0.3 | 10.9 | 430 |
| k21_NOROLL | 854 | 841 | 841 | 874 | 874 | 1.5 | 1.5 | -2.3 | -2.3 | 10 |
| k21_UNROLL2 | 901 | 848 | 774 | 899 | 837 | 5.9 | 14.1 | 0.2 | 7.1 | 14 |
| k21_UNROLL3 | 894 | 856 | 730 | 903 | 737 | 4.3 | 18.3 | -1.0 | 17.6 | 18 |
| k21_UNROLL4 | 883 | 865 | 752 | 906 | 752 | 2.0 | 14.8 | -2.6 | 14.8 | 22 |
| k22_NOROLL | 824 | 821 | 822 | 824 | 824 | 0.4 | 0.2 | 0.0 | 0.0 | 21 |
| k22_UNROLL2 | 805 | 803 | 794 | 804 | 820 | 0.2 | 1.4 | 0.1 | -1.9 | 31 |
| k22_UNROLL3 | 798 | 796 | 785 | 796 | 812 | 0.3 | 1.6 | 0.3 | -1.8 | 42 |
| k22_UNROLL4 | 811 | 810 | 793 | 812 | 814 | 0.1 | 2.2 | -0.1 | -0.4 | 53 |
| 平均 | | | | | | 2.8 | 12.0 | -0.3 | 6.5 | |

**表 3** kernel18 ループ分割実験結果
Table 3 kernel18 loop division experiment result.

| kernel18 | BCC (ms) | LOOP1 (ms) | LOOP2 (ms) | LOOP3 (ms) | LOOP 合成 (ms) | 改良度 LOOP 合成 (%) |
|---|---|---|---|---|---|---|
| NOROLL | 254 | 254 | 250 | 254 | 250 | 1.6 |
| UNROLL2 | 304 | 305 | 300 | 264 | 261 | 14.1 |
| UNROLL3 | 304 | 305 | 305 | 265 | 264 | 13.2 |
| UNROLL4 | 304 | 305 | 300 | 265 | 264 | 13.2 |

GA:仮想多重使用なしの遺伝的命令スケジューラ
GA-ST:仮想多重使用した遺伝的命令スケジューラ
list:仮想多重使用なしのリストスケジューラ
list-ST:仮想多重使用したリストスケジューラ
改良度:100-スケジュール後の実行時間/BCC*100
IS_Num:対象としたアセンブリ命令数
となっている．また，LFKNo. にある平均は，表 1 では (kernel1〜24 までの改良度の和)/24，表 2 では (UNROLL2〜4 の改良度の和)/24 となっている．

### 4.2 考　　察

表 1 では，平均をみると仮想多重使用を用いた遺伝的命令スケジューラの結果が最も優れていた．一方、リストスケジューラの結果ではスケジュール前よりも遅くなっているものが存在した．GA と GA-ST を比較すると，kernel7,9,16 において効果があったと言える．これらの命令列は他の命令列と比べ，スタックの

仮想多重使用によって，FPU 命令のスケジューリングが可能になっていた．スケジュール後の命令列では，FPU 命令が入れ替わっており，適宜，fxch 命令が挿入されているように変化していた．その他の命令列では，スタックによる依存はなくなったものの，FPU 命令が３～４個しか存在せず，FPU 命令の SDN 値がすべて等しかったり，他のレジスタやメモリによる依存があることで，今回の仮想多重使用の効果が発揮できなくなっていた．GA-ST では，kernel18 のみが悪い結果を示している．kernel18 は命令数が多く，３つのループに分けることができる．そこで，ループ全体ではなく，これらの３つのループ LOOP1～LOOP3 のみをそれぞれ対象に，遺伝的命令スケジューリングをおこなった．依存グラフは GA_ST と同じものを使用した．その結果が表 3 である．LOOP 合成とは，LOOP １～3 の部分の命令列を合わせたものである．これにより，探索範囲を狭めることで，表 1 よりも良い命令列を得ることができた．

表 2 は，表 1 よりも改良度が高くなっている．これは C ソースコードのループ部分を展開したことによって，データ依存の無い FPU 命令が増加したからである．しかしながら，ループ展開したのにも関わらず kernel22 は１～２%の改良度であった．これは，kernel22 の各ループ内に call 命令があり，関数呼び出しによる依存のため，ループ展開の効果がなくなっていたのである．また，表 2 の結果を見る限りでは，各 kernel の UNROLL2～4 は，NOROLL と比較すると，改良度に大きな差はない．したがって，展開するループ数は２で十分であると考えられる．

kernel7 をはじめ，表 1 と表 2 において，GA-ST や list-ST では改良度が大幅に上昇しているものがある．それらのスケジューリング後の命令列は異なる SDN 値の FPU 命令が混じりあっていた．それによって，おのおのの FPU 命令のレイテンシによる遅延が他の依存のない FPU 命令によって隠れたと考えられる．kernel7 だけでなく，GA-ST や list-ST において，改良度が上がっているプログラムは，スケジューリング後の命令列は同じ傾向にあった．それが改良度の向上に繋がった原因と言える．

## 5. 結　　言

今回の研究では，PentiumIII のアセンブリ命令列に対し，データスタックの仮想多重使用を用いることで，以下の事が明らかになった．

- 仮想多重使用とスタック入れ替え命令の自動挿入アルゴリズムを組みこむことで，データスタックによる命令間依存を回避し，多くの浮動小数点計算を含む Pentium アセンブリ命令列を効果的にスケジューリングできた．これは特にループ展開した命令列に対して効果的であった．
- LFK のループ展開した命令列を対象とした実験の結果，最大 26.5%の性能向上が得られた．また，ループ展開しない命令列に対しては最大 14.6%の性能向上が得られた．

今後の課題として，次の点が挙げられる．

- より大規模な問題を対象とした評価．
- 他の PC プロセッサへの適用実験．

**謝辞**　本研究を行なうにあたり，論文の草稿に貴重な御意見を寄せて頂いた静岡大学情報学部 秋川友宏氏に感謝致します．

## 参　考　文　献

1) Intel Corporation:“ インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル 上巻 ”，資料番号 243190J，(1999)
2) Umetani,Y.:“ Application of genetic algorithm to instrucion sequence optimization for risc processor. ”Technical Report838,GMD,(1995).
3) 伊東勇，梅谷征雄:“ 遺伝的アルゴリズムを用いる命令スケジューリング方式とその効果 ”, 情報処理学会論文誌,Vol. 41,No. 4,pp. 1073–1085 (2000).
4) 長倉裕叔，梅谷征雄 :“ PC アーキテクチャにおける遺伝的命令スケジューリングの適用実験 ”, 情報処理学会研究報告 Vol. 2000,No. 57,pp. 31–36(2000)
5) 中田育男: “ コンパイラの構成と最適化 ”， 朝倉書店 (1999)
6) Steven S. Muchnick:“ *Advanced* COMPILER DESIGN IMPLEMENTATION ”，MORGAN KAUFMANN, (1997)
7) McMahon， F.H.:“ The livermore fortran kernels: A computer test of numerical performance range ”， Technical report， Lawrence Livermore National Laboratory (Dec. 1986).
8) Intel Corporation:“ インテル・アーキテクチャ・ソフトウェア・ディベロッパーズ・マニュアル 中巻 ”，資料番号 243191J，(1999)
9) Steven J. Beaty: “ Genetic Algorithms and Instruction Scheduling. ”Proceedings of the 24th Annual International Symposium on Microarchitecture，(Nov. 1991)
10) 坂和正敏， 田中雅博: “ 遺伝的アルゴリズム ”, 朝倉書店 (1996)
11) M.Bekerman, A.Yoaz, F. Gabbay , S.Jourdan ,M.Kalaev , R.Ronen: “ Early Load Address Resolution Via Register Tracking ”，Proc. of the 27th ISCA, pp. 306–315(2000)