ハイパフォーマンス
コンピューティング 81－6
（2000．6．2）

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

長倉裕叔*1　梅谷征雄*2

RISC プロセッサの性能を上げるために遺伝的アルゴリズムを命令スケジューリングに応用することが提案されており，SPARC の Ultra2 プロセッサ上で GNU－C コンパイラと比べ，最大 25.8%の性能向上が確認されている．今回の実験では RISC アーキテクチャではない Pentium プロセッサ上で遺伝的命令スケジューリングを行い，その性能を評価した．その結果，Borland C++ Builder4 コンパイラと比べ，最大 5%の実行時間の短縮を確認した．また，SPARC プロセッサに対して効果の小さい要因として，FPU データレジスタがスタック構造であることや命令形式の違いが挙げられる．

Experiments of the Instruction Scheduling using Genetic Algorithm for Personal Computer's architecture

Hirotoshi Nagakura*1　Yukio Umetani*2

An Application of Genetic Algorithm to the instruction sequence optimization was proposed to squeeze out RISC processor performance, and it was already verified that the performance is higher up to 25.8% compared to the GNU-C compiler on Ultra2 processor. In this experiments, the performance has been evaluated on a Pentium processor. As a result, we verified that the performance is higher performance up to 5% compared to the BorlandC++Builder4 compiler. The causes that Pentium processor shows less performance up are reasoned as the FPU data registers are treated as a register stack, and the format instruction is different.

1．まえがき

RISC プロセッサの場合，命令スケジューリングによってプログラムの実行時間が変化する．つまり，プロセッサのスループットを向上するには，パイプライン処理の遅延を減らす命令スケジューリングが重要となる．しかし，従来の最適化コンパイラでは依存する命令のペアを引き離すだけの簡単なスケジュール規則を使っているだけである．これでは急速にパイプライン制御方式を複雑化している RISC プロセッサに十分適した最適化が行われていない可能性がある．

このような現状を改善するために，詳細なマシン仕様に依存しない新しい最適化法，すなわち，遺伝的アルゴリズム（genetic algorithm：GA）を用いる方法が導入された[1]．

遺伝的アルゴリズムとは，適用範囲の非常に広い，生物の進化を模倣した学習的アルゴリズムであり，何億年もかけて生物が進化してきた遺伝的な法則を工学的にモデル化し，また参考して工学に役立つ学習方法を与えるものである[2]．

これまで，遺伝的アルゴリズムを使用した命令スケジューリングを SPARC の Ultra2 に応用した実験が行われている[3][4]．この実験では遺伝的アルゴリズムを基に遺伝的命令スケジューラを作成し，そのスケジューラと GNU－C コンパイラによってそれぞれ得られた命令列の実行時間を比較した結果，最大 25.8%実行時間を短縮することに成功している．さらに，その遺伝的命令スケジューラは，対象となるプログラムの実行時間にのみ依存したスケジューリングをするため，マシン特性に適応した効果が得られる事が示された．

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

RISC 以外のアーキテクチャで遺伝的命令スケジューラがどのような効果を示すかは興味があるところである．そこで，今回の実験は，PC アーキテクチャの代表的である Intel の Pentium に適用して評価を行った．

2. 遺伝的命令スケジューラ

2.1 遺伝的アルゴリズム

遺伝的命令スケジューラのアルゴリズムを図1に示す．

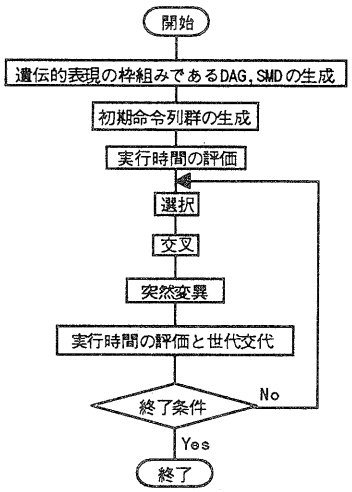

図1：遺伝的命令スケジューラのアルゴリズム

また，このアルゴリズムを以下の〔1〕～〔8〕で詳しく説明する．

〔1〕遺伝的表現の枠組みである DAG，SMD の生成
対象とするアセンブリ命令列を基に，DAG や SMD を生成する．

〔2〕初期命令列群の生成
あらかじめ変数 *NPOPL* を設定しておき，SMD を基に，*NPOPL* 個の初期命令列を生成する．

〔3〕実行時間の評価
初期命令列をそれぞれ実際にマシン上で実行し，各命令列の実行時間を得る．

〔4〕選択
NPOPL/2 個の命令列を実行時間の短い順に選択していく．

〔5〕交叉
選択した命令列群からランダムに *NPOPL/2* 個の対を作り，これらを交叉し新たな SMD を作成することで *NPOPL* 個の子孫を生成する．

〔6〕突然変異
各命令列に対して，確率的に突然変異をさせる．これによって SMD が変化する．

〔7〕実行時間の評価と世代交代
NPOPL 個の命令列をそれぞれ実行し，実行時間を計測する．その結果，前の世代までの最小実行時間よりも短い実行時間を持つ命令列が無ければ，その最小実行時間を持つ命令列と今回生成した命令列の中の1つとが置き換わる．このようにして得られた *NPOPL* 個の命令列が次世代の個体になる．

〔8〕終了条件
あらかじめ設定した世代交代数といった終了条件を満たせば，そのときに得られている最良の命令列を目的の命令列として終了する．

2.2 遺伝的表現

命令列の遺伝的表現には DAG と SMD を用いている．

DAG (Dependence Analysis Graph)

DAG とは命令間の依存による実行順序の関係を示したグラフであり，命令スケジューリングに対する制限を示す．したがって，DAG によって，それぞれの命令の移動可能である範囲を知ることができる．この命令の移動可能範囲の決定するのに必要な情報が命令間の依存関係である．この依存関係にはレジスタやコンディションコードなどによるデータ依存と分岐などによる制御依存がある．

例として LFK（Livermore Fortran Kernels）2 の命令列の一部を使って説明する．その命令列を図2 に示す．また，図2の DAG を図3に示す．

図3では命令間の依存関係をアークで示している．また，DAG の先頭から依存アークを順に辿り，部分命令列を切り出すことができる．この部分命令列をストリングと定義している．この作業を繰り返すことによって，命令列をストリングの集合に分ける．図3では太線で囲まれた部分がそれぞれストリング

である．そして，ストリングの集合において，最長のストリングをクリティカルパスとしている．

DAG生成の改良

今回の実験対象にしているPentiumではadd命令などの整数算術命令を実行するとフラグレジスタに書き込みをする[5]．したがって，整数算術命令を多く使用している命令列では不必要な書き込みによるフラグ依存が多くなると予測できる．

```
1 @15:
2        fld     qword ptr [edx]
3        fmul    qword ptr [eax-8]
4        add     ebx, 8
5        add     ecx, 2
6        fsubr   qword ptr [eax]
7        fld     qword ptr [edx+8]
8        fmul    qword ptr [eax+8]
9        add     eax, 16
10       add     edx, 16
11       cmp     esi, ecx
12       fsubp    st(1), st
13       fstp    qword ptr [ebx]
14       jg      short @15
```

図2：kernel2のアセンブリ命令列の一部

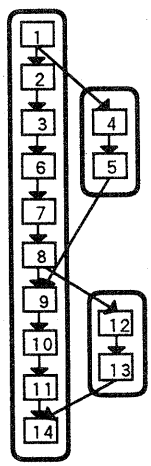

図3：図2のDAG

そのため，フラグに関する依存関係を改善することで命令順序の入れ換えパターンが以前よりも多いDAGを生成できるように改良した．

例えば，同じフラグに書きこむ命令が連続して存在していると仮定する．この場合，最初と最後の命令の間にある命令は互いに命令順序をその範囲内で交換しても影響はない．このような方法で図2のコードのDAGを改良すると，図4のDAGになる．

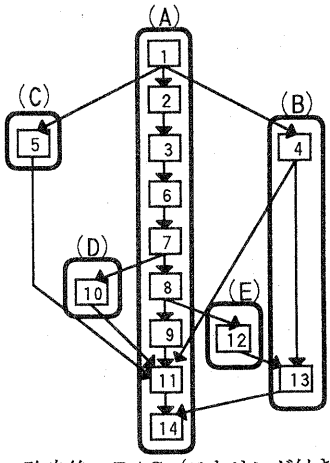

図4：改良後のDAG（ストリング付き）

SMD (String Merge Diagram)

各ストリングの親ストリング（マージ対象ストリング）へのマージ方法とストリングに対するマージ順を決めれば一本化された命令列が決定できる．SMDは命令列全体の組み立てをマージの階層に分解して示す枠組みである．

SMDでは各ストリングにレベルが存在している．そのレベルは以下の手順〔1〕～〔4〕で設定される．

〔1〕ストリングaとストリングbをそれぞれ構成する命令間に1つ以上の依存関係があるとき，aとbは相互に依存すると定義する．

〔2〕全ストリングをレベル0として初期化する．

〔3〕最長のストリング(クリティカルパス)と，これに依存しないストリングをレベル1とする．

〔4〕レベルi(i=0，1，2…，n)以下のストリングにのみ依存するストリングを選び，レベルを(i+1)　とする．すべてのストリングがレベル付けされるまで，これを繰り返す．

SMDはレベル付けしたストリングにSMS

(String Merge Status) と SMO (String Merge Order) を加えて構成されている．まず，SMS はレベル2以上のストリングに存在している各命令を，それより低いレベルのストリングのどの命令後に挿入するべきかを示しているコードである．そして，SMO はストリングをどのような順序でマージするかを示すものである．図4から作成できる SMD の一例を図5に示す．

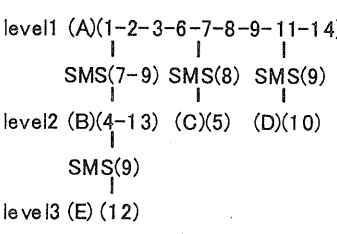

図5：図4のSMD

図5において，level2 であるストリング(B)の命令(4)(13)の SMS コードがそれぞれ(7)(9)となっている．これはストリング(B)の命令(4)(13)がストリング(A)の命令(7)(9)の後に挿入されることを示している．さらに同レベルのストリングが多数存在する時には，それぞれのストリングをマージする順序が必要になる．それを示しているのが SMO である．図5では(A)-(B)-(D)-(C)-(E)の順序になっている．

したがって，図5の SMD から得られる命令列は(1)(2)(3)(6)(7)(4)(8)(5)(9)(12)(10)(13)(11)(14)となる．

2.4 遺伝的操作

遺伝的命令スケジューラで行う遺伝的操作について述べる．詳しくは参考文献 3) 4)を参照願いたい．

2.4.1 初期命令列群の生成

SMS と SMO を決定し，命令列を生成する．SMS は DAG の範囲内でランダムに決定する．また，低レベルのストリングから，ストリング内の命令は先頭から順に決定する．SMO は同レベルのストリングをランダムに交換して決定する．

2.4.2 交叉

SMS 又は SMO の一部を対の命令個体の間で交換する．SMS ではランダムに選択したストリングの SMS コードを交換する．しかし，交換の対象となる SMS コードが同じならば交換しない．また，SMO の場合は選択されたレベルの SMO コードをすべて交換する．SMS と同様，SMO が同じであれば交換しない．

2.4.3 突然変異

ランダムに選択した SMS コードの一部または選択したストリングレベルの SMO の一部を確率的に変える．

3． LFK による性能評価

ここでは LFK ベンチマーク [6] を利用して Borland C++ Builder4 コンパイラが生成するアセンブリコードを対象に遺伝的命令スケジューリングを行い，その性能を評価する．今回の実験で使用したマシンは GATEWAY G6−450（メモリ 128MB）であり，Pentium Ⅱ 450 を搭載している．

3.1 実験方法

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，それを 24 個のプログラムに分割し，さらに FORTRAN から C に書き直したもの(kernel1～24)を使用した．

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．この関数を使い，それぞれのカーネルプログラムにあるループについて，そのループの開始から終了までにかかる時間を計測した．

また，遺伝的命令スケジューラでの初期命令列群の数を 10，世代交代数を 30 世代とした．

実験手順を示すと以下の〔1〕～〔4〕となる．

〔1〕 実験対象のプログラムを最適化コンパイルし，アセンブリコードを作成する．つまり，ここでは Borland C++Builder4 コンパイラを使って，カーネルプログラムを最適化コンパイルする．

その際，できるだけ高速のコードを生成するコンパイラオプション O2 を使用した．

〔2〕〔1〕で作成したコード中のループ部分の命令列を対象に遺伝的命令スケジューリングをする．その結果，出力として準最適なループ部分の命令列を持つコードを得る．

〔3〕〔2〕で得たコードをアセンブルし，実行ファイルを作成する．また，〔1〕で作成したコード，つまり遺伝的命令スケジューリングをしていないコードの実行ファイルも作成する．

〔4〕2つの実行ファイルを実際に実行し，それらの実行時間を比較する．

3.2 実験結果

表1に Borland C++ Builder4 コンパイラと遺伝的命令スケジューラのコードによる実行時間の結果をのせる．それぞれの実験結果は 10 回の測定値の平均をとっている．表1において，

LFKNo. :kernel番号
GA1:遺伝的命令スケジューリングあり (DAG改良前)
GA2:遺伝的命令スケジューリングあり (DAG改良後)
BCC：Borland C++ Builder4コンパイラのみ
改良度1:100 - GA1/BCC×100 (%)
改良度2:100 - GA2/BCC×100 (%)

となっている．つまり，この改良度が0％より高ければ，遺伝的命令スケジューリングの効果があったことになる．また，Ultra2 改良度は参考文献 3) 4) から参照したものであり，これは GNU−C コンパイラと比較した改良度になっている．

3.3 考察

表1を見ると，改良度1・2では改良度が0〜5%である．したがって，Borland C++ Builder4 コンパイラだけのコードよりも遺伝的命令スケジューリングを施したコードの方が優れているといえる．

今回の実験で最も改良度が高かった kernel2 のコードを検討した結果，頻繁に使用されているレジスタに依存している命令間にはさみこまれている命令数が均等化されている傾向があった．このように，パイプラインを有効に活用する命令列が作成されたことが改良度の高くなった原因と思われる．

そして，比較対象としているコンパイラは違うが，Pentium と Ultra2 の改良度に大きな違いが生じている原因として次のことが考えられる．

まず，命令数の違いである．例えば下のような Pentium のアセンブリ命令があったとする．

```
・fmul   qword ptr [eax - 8]
```

この命令と同じ動作を SPARC のアセンブリ命令で記述すると次のようになる．

```
・ldd    [%g3+%o4] , %f2
  fmuld   %f4 , %f2 , %f4
```

このように SPARC は，レジスタのみで浮動小数点演算をするため，メモリ上の値を必要とする時はその値をレジスタにロードしなければならない．しかし，この2命令を分離することにより，入れ換えの可能性を多くできる．さらに，Pentium は SPARC と違って，FPU データレジスタをスタックレジスタとして扱っている．そのため，浮動小数点を扱う命令間の依存が強く，それらの命令順序は入れ換えにくいのである．今回使用した LFK は浮動小数点演

表1 実験結果と遺伝的スケジューラによる改良度

LFK No.	GA1 (ms)	GA2 (ms)	BCC (ms)	改良度1 (%)	改良度2 (%)	Ultra2 改良度 (%)
1	3302	3291	3311	0.3	0.6	25.8
2	1271	1260	1327	4.2	5.0	19.2
3	5508	5506	5511	0.1	0.1	0.6
4	840	839	869	3.3	3.5	9.2
5	1099	1087	1109	0.9	2.0	13.4
6	487	489	496	1.8	1.4	5.2
7	1981	1966	1988	0.4	1.1	22.2
8	853	850	860	0.8	1.2	9.8
9	2784	2785	2807	0.8	0.8	16.6
10	4782	4693	4872	1.8	3.7	2.2
11	546	546	546	0	0	15.5
12	570	570	570	0	0	14.2
13	502	504	512	2.0	1.6	8.0
14	1368	1370	1401	2.4	2.2	12.6
15	1484	1483	1485	0.1	0.1	3.2
16	620	619	621	0.2	0.3	7.5
17	1950	1997	2026	3.8	1.4	8.4
18	539	545	547	1.5	0.4	5.9
19	1456	1452	1463	0.5	0.8	9.0
20	1639	1638	1648	0.5	0.6	7.3
21	2369	2354	2371	0.1	0.7	4.4
22	1484	1483	1487	0.2	0.3	0.2
23	1093	1089	1113	1.8	2.2	18.5
24	988	988	990	0.2	0.2	0.4

算が多い．したがって，SPARC よりも Pentium は，命令数が少なく，命令順序も入れ換えにくい．これが，Pentium と SPARC の改良度の違いが生じた大きな原因であろう．

また，他の原因としては，Borland C++ Builder4 コンパイラの最適化によって得られたコードが最適命令列に近いものであったことが考えられる．

それから，改良度 1 と 2 を比較すると，改良度 2 の方が改良度の高い kernel が多い．つまり，DAG 生成の改良によって，改良度を上げることができたのである．これは DAG 生成の改良によって，命令順序を入れ換えられるパターン数が増えたことで，より優れた命令列パターンが生成できるようになったためである．しかし，逆に改良度 1 の方が改良度の高い場合がある．これは改良によって優れた命令列パターンが生成できず，命令順序の入れ換えパターンのみが増える結果になってしまい，改良前は容易に得られた準最適命令列を得るのが難しくなったからであろう．

4． 結論

今回の研究では遺伝的命令スケジューラを Borland C++ Builder4 コンパイラと比較して以下のことが明らかになった．

- 遺伝的命令スケジューラの適用によって，Borland C++Builder4 コンパイラに比べ，最大 5%の実行時間の短縮が確認できた．つまり，Pentium において遺伝的命令スケジューラは有効である．

- 実行時間の短縮の要因はレジスタの使用待ちの遅延をうまく解消していることにある．したがって，遺伝的命令スケジューラを使うことで，パイプラインをより有効に活用する命令列が作成できる．

- SPARC に比べて，遺伝的命令スケジューラの効果が小さい．

- Pentium の場合，フラグ書きこみの依存を考慮した DAG 生成の改良をすることで，遺伝的命令スケジューラによる改良度が増すことが多い．

また，次のような研究課題が明らかになった．

- 今回の DAG 生成の改良によって，改良前の改良度の方が高くなる不具合を修正する．

- 遺伝的命令スケジューリングにソフトウェアパイプライニングを組み込む．

- 遺伝的命令スケジューリング結果を応用し，新たな命令スケジューリングのアルゴリズムを考案する．

- スケジューリング時間の短縮と性能向上する．

これらの研究課題を考慮して，今後の研究を進めていきたい．

謝辞

本研究は平成 12・13 年度文部省科学研究費補助金 10878046「遺伝的アルゴリズムを用いる適応型命令スケジューリングの研究」により実施したものである．

参考文献

1)Yukio Umetani: "Application of Genetic Algorithm to Instruction Sequence Optimization for RISC Processor". Technical Report 838, GMD, 1995

2)坂和正敏　田中雅博:「遺伝的アルゴリズム」第 3 版，朝倉書店，1997.

3)伊東勇　梅谷征雄：「遺伝的アルゴリズムを用いた命令スケジューリングの試み」情報処理学会研究会 HPC74-12(1998.12.11)

4)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　第 41 巻 4 号(H12 年 4 月)

5)インテル・アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル中巻：命令セット・リファレンス　ftp://download.intel.co.jp/jp/developer/jpdoc/24319202_j.pdf

6)PHASE － Parallel and HPC Application Software Exchange　http://phase.ctl.go.jp/