ハイパフォーマンス
コンピューティング 74−12
（1998. 12. 11）

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

伊東 勇[†]　梅谷 征雄[††]

RISC プロセッサの性能をあ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を提案する．これにより,GNU-C コンパイラによるスケジューリングよりも高い性能が得られて，またマシンに依存しない安定した最適化が期待できる．Livermore の 24 個のカーネルに対して遺伝的命令スケジューリングを 3 つの UNIX マシン上 (Ultra2, Ultra1, SPARC) で行い，その性能を評価して最大 25.9% の性能向上を得た．またこれより，従来のスケジューリング方式が古いマシン特性に依存し，またレジスタによるデータ依存のオーバーヘッドを考慮していないことを明らかにした．

Experiments of the Instruction Scheduling using Genetic Algorithm

Isamu Ito[†]　Yukio Umetani[††]

An Application of Genetic Algorithm to the instruction sequence optimization is proposed to squeeze out RISC processor performance. By this way, far better results than those obtained by the scheduling of GNU-C compiler are obtained, and a stable optimization independent of the features of the machine is attained. The performance has been evaluated using 24 Livermore kernels on 3 UNIX machines(Ultra2, Ultra1, SPARC), and higher performance up to 25.9% compared to the conventional scheduling is obtained. We also point out that conventional scheduling methods depend on a specific machine feature, and doesn't care for the date dependence by registers.

1　まえがき

RISC プロセッサの性能は処理される命令の順序に強く依存する．ゆえに，命令間の依存関係を維持しながらプロセッサの性能をあげるために命令をスケジューリングすることは非常に重要な役割を果たす．しかし，従来の最適化コンパイラでは各々のプロセッサの特性を具体化する経験的な規則によって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なので，プロセッサ性能を十分に発揮することができない．さらには RISC プロセッサの急速な進化にコンパイラ技術がついていけなくなっている．

この現状を打破するために筆者達は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) を用いる方式を考案した [1]．GA は 1960 年代の終わりから 1970 年代の初めに,John H. Holland とその同僚やミシガン大学の学生らによって，自然界のシステムの適応過程を説明し，生物の進化のメカニズムを模擬する人工モデルとして提唱されてきた手法である [3]．その原理は，突然変異や個体の交叉により遺伝子の多数の組合せが偶然の要素をもって発生し，それがたまたま環境によく適合すれば増殖し，そうでなければ消滅するという進化法則を工学的にモデル化したものである．

GA の特徴としては次のようなものがある [4]．

1. 一つの点 (個体) から他の点へと探索を進めるのではなく，点の集合 (個体群) から集合へと探索を進めるので，初期値に比較的に依存しにくい．
2. 適応度を利用するだけで他の (勾配などの) 情報を使わないので，目的関数を性質がよくわからないような問題についても適用できる．
3. 確率的な遷移ルールに従って挙動するので，局所最大点にとどまらずに大域的な最大点に到達し得る可能性が高い．

これまでに GA は巡回セールスマン問題やスケジューリングやナップサック問題のような離散的な組合せ問題を含む多くのアプリケーションに応用されてきたが，コンパイラに関連した分野では研究が進

[†]静岡大学 大学院 理工学研究科 計算機工学専攻
Department of Computer Sciences, Faculty of Science and Engineering, Shizuoka University
[††]静岡大学 情報学部 情報科学科
Department of Information Sciences, Faculty of Information, Shizuoka University

んでいない．ゆえに命令スケジューリングにGAを適用してみることはたいへん興味深いことである．

実際のコンパイラの最適化では命令スケジューリングの他にレジスタ割当てやデータ割当て，高度なコード最適化などの多くの要素が組合さっている．しかし，この研究では命令スケジューリングのみに焦点を絞って取り扱った．

GAを命令スケジューリングに応用して実行時間という直接的な評価尺度にのみ依存したスケジューリングをすることにより，マシン特性に応じた安定した効果を期待できる．

本論文の構成は次のようになっている．2章で遺伝的スケジューラの構成を示し，3章で遺伝的スケジューラの評価をする．そして最後に，4章で今後の研究とともに結論を述べる．

2 遺伝的命令スケジューラ

2.1 アルゴリズム

遺伝的アルゴリズムを適用した遺伝的スケジューラのアルゴリズムを図1に示す．手順としてはまず命令

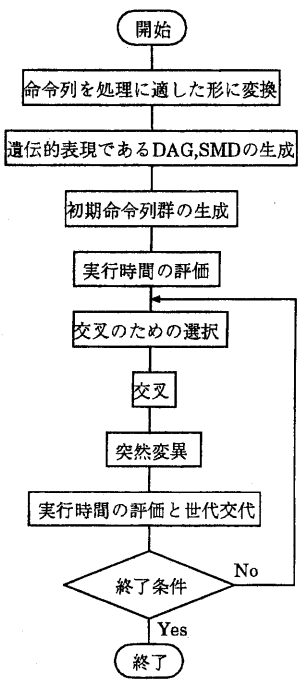

図 1: 遺伝的スケジューラのアルゴリズム

列をプログラム内での処理に適した形に変換してから

```
1   .LL19:
2           ldd     [%o4+%o2], %f2
3           sll     %o3, 3, %o0
4           ldd     [%g3+%o0], %f4
5           faddd   %f2, %f4, %f2
6           add     %o3, 1, %o3
7           cmp     %o3, 1000
8           add     %o2, 8, %o2
9           ble     .LL19
10          std     %f2, [%o4+%o0]
```

図 2: Kernel11のアセンブリーコードの一部

遺伝的表現であるDAGとSMDを生成する．そしてこれをもとにNPOPL個の初期命令列群を生成する．これらの各々の命令列の実行時間を評価し，この中から実行時間の適合値の高い順にNPOPL/2個の命令候補群を取り出す．実行時間の評価はマシン上の実測により行う．そしてこの命令列群からランダムにNPOPL/2個の対を選んで交叉をさせ，NPOPL個の子孫を形成する．次にこのNPOPL個の命令列群に確率的に突然変異をさせる．この遺伝的操作の後に実行時間を評価して，世代交代する．世代交代では古い世代でも最小の実行時間を持つものは次世代に引き継ぐ．そして再びこの新世代の命令列をもとに選択して遺伝的操作を行う．この操作を終了条件(今回はあらかじめ設定した世代交代数)を満たすまで繰り返す．これによって生成された命令列群中最小の実行時間を持つものが，準最適な命令列である．

2.2 遺伝的表現

命令列の遺伝的表現はDAGとSMDを用いて作成する．

DAG(Dependence Analysis Graph)とストリング

DAGは命令の依存間の実行順序に関する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．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やディレイスロットなどによる制御依存がある．

例として，LFK(Livermore Fortran Kernels)11の命令列の一部を使って説明する．図2にアセンブリーコード，図3にDAGを示す．命令間の依存関係がアークで示されている．このDAGの先頭から依存アークを優先順に辿ることによって部分命令列を切り出すこ

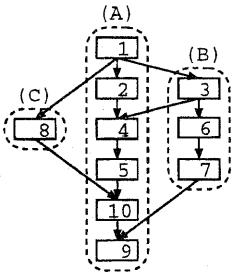

図 3: 図 2のコードの DAG

とができる. この部分列をストリングと呼ぶ. この切り出しを繰り返すことにより命令列をストリングの集合に分けることができる. 図 3では点線で囲まれた部分がストリングである. それぞれのストリングは相互の依存関係を保つかぎり並列実行が可能である. DAG はクリティカルパスという最長のストリングを含んでいる.

SMD(String Merge Diagram)

ストリングは次の方法でレベル付けを行うことができる. 最長のストリングとこれに依存しないストリングをレベル 1 とする. そして以降レベル i 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 (i+1) とし, これを全てのストリングがレベル付けされるまで繰り返す. 図 3では最初に切り出したストリング (A) にレベル 1, 続いて切り出した (B) と (C) にレベル 2 がつく.

SMD は DAG の制約を満たして生成可能な命令列のうちの確定した一つを示す図式であり, SMS(String Merge Status) と SMO(String Merge Order) を含む. SMS はストリングを構成する各命令が, 下位レベルのストリングのどの命令の後に挿入されるべきかを示し, SMO はストリングをマージする順序を示すものである.

例として, 図 4に図 3に対する SMD を示す. 図 4で

```
lebel1 (A)(1-2-4-5-10-9)
          |              |
       SMS(2-5-5)   SMS(5)
          |              |
lebel2 (B)(3-6-7)   (C)(8)

SMO: (A)-(C)-(B)
```

図 4: 図 3の SMD

はレベル 2 の命令 (3)(6)(7) の SMS コードがそれぞれ (2)(5)(5) となっており, これはそれぞれの命令がストリング (A) の命令 (2)(5)(5) の後にこの順に挿入されることを表す. さらに同じレベルのストリングが多数存在するときなどには, それぞれのストリングのマージする順番が必要となる. これを決めるのが SMO である. 図 4では (A)-(C)-(B) の順序になっている. 図 4の SMD で生成される命令列は (1)(2)(3)(4)(5)(6)(7)(8)(10)(9) である. その後, ディレイスロットの命令 (10) とその直前の命令 (9) を入れ換えて最終的な命令列とする.

2.3 遺伝的操作

遺伝的スケジューラで使う遺伝的操作を SMD を用いて以下のように定義する.

1. 初期命令列群の生成
   SMS と SMO を決定して命令列を生成する. SMS は DAG の範囲内でランダムに決定する. 順序としては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. 交叉
   SMS または 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. SMO では選択されたレベルの SMO を交換する. それぞれ条件があり,SMS では交換の対象となる SMS コードが両方で異なり, さらに対象とするストリングより低いレベルのストリング群をマージして作る部分命令シーケンスが両方で同じであることである. SMO では選択されたレベルの SMO が両方で異なることである.

   図 4と図 5に示す SMD の交叉の例を示す. ま

```
lebel1 (A)(1-2-4-5-10-9)
             |          |
          SMS(1-4-10) SMS(1)
             |          |
lebel2 (B)(3-6-7)  (C)(8)

SMO: (A)-(B)-(C)
```

図 5: 図 3の SMD

ず, これらの SMD が生成する命令列は

- (1)(2)(3)(4)(5)(6)(7)(8)(10)(9)
- (1)(8)(3)(2)(4)(6)(5)(10)(7)(9)

である. ここでストリング (B) の SMS, すなわち (2-5-5) と (1-4-10) を互いに交換するとそれぞれの命令列は,

- (1)**(3)**(2)(4)**(6)**(5)(8)(10)**(7)**(9)
- (1)(8)(2)**(3)**(4)(5)**(6)(7)**(10)(9)

となる. また,SMO でレベル 2 のストリング, すなわち (C)-(B) と (B)-(C) を交換すると,

- (1)(2)**(3)**(4)(5)**(8)(6)(7)**(10)(9)
- (1)**(3)(8)**(2)(4)**(6)**(5)(10)**(7)**(9)

となる.

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.

   図 5においてストリング (B) の (6) の SMS コードが (4) から (2) に変異したとすると命令列は

   - (1)(8)(3)(2)**(6)**(4)(5)(10)(7)(9)

   となる. また,SMO のレベル 2 の部分の (B) と (C) が変異によって交換されると

   - (1)**(3)(8)**(2)(4)**(6)**(5)(10)**(7)**(9)

   となる.

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 GNU-C コンパイラ gcc2.8.1 が生成するアセンブリーコードを対象に遺伝的スケジューリングを行い, その性能を評価する. マシンの特性による変化に着目するために 3 つの UNIX マシン Ultra2(300MHz),Ultr-a1(1-43MHz),SPARC(100MHz) で実験を行う.

3.1 実験の方法

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おり, 計算機の実行速度を比較するベンチマークプログラムとして使われている. 今回の実験では筆者は取り扱いやすいように 24 個の独立プログラムに分割し, さらに GNU-C コンパイラでのスケジューリングと比較するために FOTRAN から C に書き直した.

遺伝的スケジューラでの実行時間の評価としては UNIX の組み込み関数である clock を使用した. clock はマニュアルによると, ユーザ時間とシステム時間の合計の CPU 時間をマイクロセカンドのオーダーで計測する. しかし実際には,$10ms$ が分解能の限界である.

それぞれのプログラムにおけるループ回数は, 筆者が clock による誤差と計測時間を考慮して独自に設定した. Ultra1,2 と SPARC ではマシン性能があまりにもかけ離れていたためにループ回数は Ultra1 と 2 を同じに,SPARC はその 1/10 に設定した. また, 今回は準最適命令列の生成までの時間があまり長くなりすぎないように遺伝的スケジューラでの初期命令群の数を 10, 世代交代数を 10 世代とした.

おおまかな手順は次のようになる.

1. それぞれのカーネルを最大の最適化オプション O3 を指定してコンパイルし, アセンブリーコードを生成する. このオプションを指定すると, 様々な最適化とともに M.D.Ticmann[2] によるスケジューリングが行われる.

2. このアセンブリーコードのループ部分を中心とした遺伝的スケジュール対象部分を独自のツールを使って遺伝的スケジューラ内での処理に適した仮想 RISC プロセッサ言語に変換する.

3. アセンブリー言語ファイル, スケジュール対象部分の仮想 RISC 言語ファイルとともに遺伝的スケジューラを実行する. 出力としては, 進化の過程と準最適な命令列が得られる.

4. この操作を 3 つのマシンで 24 個のカーネルに対して行う.

3.2 結果

表 1に GNU-C コンパイラと遺伝的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 clock による誤差のために 20 回の測定値の平均をとっている. 各々のマシンについて左からコンパイラによる命令列の測定値, 遺伝的スケジューラによる準最適命令列の測定値を μs の単位で, またコンパイラのスケジューラに対する改良度を % の単位でのせている. 一番右側には参考のために遺伝的スケジューラの対象とした範囲の命令数をのせている.

表 1: 機種による改良度の比較

NAME	Ultra2			Ultra1			SPARC			IS-Amo
	Copt-Exc(μs)	Sopt-Exc(μs)	S/C* 100(%)	Copt-Exc(μs)	Sopt-Exc(μs)	S/C* 100(%)	Copt-Exc(μs)	Sopt-Exc(μs)	S/C* 100(%)	
kernel1	1632500	1211500	74.2	3405000	2561500	75.2	2970000	2973500	100.1	49
kernel2	1073500	867000	80.8	2252000	1802000	80.0	2740000	2691000	98.2	54
kernel3	1023500	1017500	99.4	2130500	2125500	99.8	1230000	1229000	99.9	14
kernel4	1473500	1337500	90.8	2942000	2652000	90.1	3098500	3098000	100.0	42
kernel5	1594000	1381000	86.6	2991500	2571000	85.9	2040500	1940000	94.9	33
kernel6	1102500	935000	94.8	2045500	1757500	85.9	2111300	1945000	92.1	48
kernel7	1538000	1196500	77.8	3022500	2286500	75.6	3325500	3113500	93.6	77
kernel8	968500	874000	90.2	1764000	1637000	92.8	1875500	1838000	98.0	177
kernel9	1543000	1286500	83.4	2939500	2415500	82.2	2881500	2983000	103.5	79
kernel10	1731000	1692500	97.8	3080500	3024500	98.2	4760500	4711500	99.0	52
kernel11	1308500	1105500	84.5	2755500	2339500	84.9	1570000	1572500	100.2	30
kernel12	1171500	1005000	85.8	2479000	2124500	85.7	2788000	2784000	99.9	28
kernel13	1044000	96000	92.0	1968000	1741500	88.5	1221000	1207500	98.9	103
kernel14	1229000	1074000	87.4	2131000	1942500	91.2	1431500	1408500	98.4	149
kernel15	1337000	1294000	96.8	2716000	2614000	96.2	1941500	1938000	99.8	228
kernel16	866000	801000	92.5	1808500	1674000	92.6	1546500	1532000	99.1	135
kernel17	1320500	1210000	91.6	2758000	2555500	92.7	2598500	2521000	97.0	77
kernel18	1232500	1160000	94.1	2306000	2188000	94.9	1855500	1949000	105.0	242
kernel19	1176500	1071000	91.0	2473000	2251000	91.0	2073500	2017000	97.3	46
kernel20	1583000	1468000	92.7	2986500	2921000	97.8	1852500	1858500	100.3	102
kernel21	1494000	1428000	95.6	2615000	2535500	97.0	3117000	3111000	99.8	45
kernel22	1439500	1437000	99.8	3007500	3004500	99.9	2173000	2159000	99.4	55
kernel23	2160000	1760000	81.5	4123500	3054500	74.1	3119000	3180000	102.0	80
kernel24	1070000	1066000	99.6	2426500	2419000	99.7	1910000	1908000	100.0	34

3.3 考察

実験結果より Ultra2,1 では全てのケースで実行時間が良くなっているが, SPARC ではほとんど改良が見られず, むしろ悪いものもある. これには 1 つの原因として今回の実験では初期命令群の数と世代交代数を実験の効率化の観点から少なく設定したことがあげられる. これを確認するために 100% を超えるものについては適当に初期命令群の数と世代交代数を増加させて実験を行ってみた. この結果, 全てのケースで GNU-C コンパイラのスケジューリングよりは良い結果を得ることができた. しかし, 他のマシンほどの改良度は得られなかった. このことから次のように考えることができる. まず,GNU-C のスケジューラの実装に関する文献 [2] は 1989 年に発表されていて, スケジューラはこの頃に最新であったマシンを使って作られていると考えられる. 実際に文献 [2] では Sun4/110 で性能評価を行っている. そして,SPARC の製造年は 1993 年であり,3 つのマシンのうちではスケジューラが GNU-C コンパイラに実装された時期に一番近い. ゆえに SPARC で生成されたコードは最適命令列により近いものであると考えられる. したがって遺伝的スケジューラでの大幅な改良度が得られず, また良い実行時間を得るまでに多くの世代交代と適度な初期命令群を要したと考えられる. このことはまた,GNU-C コンパイラのスケジューラが Sun4/110 のマシン特性に強く依存するということを意味する.

Ultra2,1 の改良度の高いものについてその要因を得られたアセンブリーコードを基に解析してみた. ここでは少ない命令数で比較的高い改良度が得られた Kernel11 のコードを例に説明を進めていく. 先にのせた図 2は Kernel11 の核となるループ部分であり, 図 6は その DAG でデータ依存を太線で示している. ここで

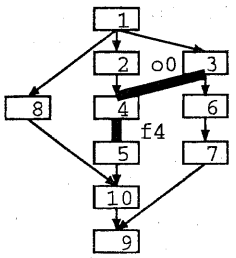

図 6: Kernel11 の DAG

表 2: 機種による遺伝的コードの違い

NAME	Ultra2		Ultra1	
	U2-code	U1-code	U1-code	U2-code
kernel7	1196500	1206000	2286500	2416500
kernel9	1286500	1309500	2415500	2509000
kernel23	1760000	1795500	3054500	3097500

遺伝的スケジューラで生成された命令列の核となる部分を以下に示す.

- Copt (1)(2)**(3)(4)(5)**(6)(7)(8)(9)(10)
- Ultra2 (1)(2)**(3)**(6)(7)**(4)**(8)**(5)**(9)(10)
- Ultra1 (1)(2)**(3)**(6)**(4)**(8)(7)**(5)**(9)(10)

このコードをDAGと照らし合わせて解析してみると, 遺伝的スケジューラはデータ依存命令をうまく引き離すようにスケジューリングしていることが分かる. ここでのデータ依存は2つの命令間の同じレジスタに関しての使用待ちで(READ,WRITE),(W,R),(W,W)の3つのケースがある. 具体的にはKernel13の(3)と(4)のo0レジスタと(4)と(5)のf4レジスタの同じレジスタの使用による遅延をうまく利用するように命令が挿入されている. この傾向は他のカーネルでも同様で改良度の大きいものはこの同じレジスタ使用の引き離しが沢山組み合わさっている.

またUltra2,1で得られたコードをそれぞれマシンを交換して実行してみた. 表2にその結果をのせる. 表1と比較してみると, それぞれのマシンでの準最適コードがマシンを替えると, 必ずしも良い結果をもたらしていないことに気付く. これから, 遺伝的スケジューラが個々のマシン特性を考慮したスケジューリングを提供していると言える.

4 結論

今回の研究では遺伝的スケジューラをGNU-Cコンパイラのスケジューラと比較して以下のようなことが明らかになった.

- LFKを対象とした実験の結果, 最大25.9%の性能向上が得られた.
- 性能向上の要因は2つの命令間のレジスタの使用待ちをスケジュールによってうまく解消していることにある.
- GNU-Cコンパイラによるスケジュールは開発当時のマシン特性に強く依存していて, マシンが急速に発展している現在では適さなくなってきている.
- 遺伝的スケジューラは実行時間という尺度のみで進化していくので, マシン特性に依存しない安定したスケジューリングが期待できる.

今後の課題としては, レジスタの使用待ちを解消するスケジューリングに重点をおいて新しいスケジュールのアルゴリズムの提案, 実装を目指して研究を進める予定である.

謝辞

本研究は平成10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 Instrucion Sequence Optimization for RISC Processor", Arbeitspapiere der GMD 838,1995.

[2] M.D. Tiemann: "The GNU instrucion scheduler" Free Software Foundation, Cambridge MA, June 1989.

[3] 坂和正敏 田中雅博：「遺伝的アルゴリズム」第2版, 朝倉書店,1996.

[4] 北野宏明：「遺伝的アルゴリズム」, 産業図書,1993.

[5] 関口智嗣 小柳義夫: 「スーパーコンピュータの性能評価の現状」, 応用数理,VOL.3 NO.1 pp27-37,1993

[6] http://phase.etl.go.jp/, PHASE - Parallel and HPC Application Software Exchange.