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

伊東 勇[†] 梅谷征雄[††]

近年のRISCプロセッサの進化，複雑化とともにその潜在的な性能を引き出す手段である，命令スケジューリングが非常に重要な役割を果たす．このため，本研究ではこのRISCプロセッサの性能を引き出す汎用的な方法として，遺伝的アルゴリズムを命令スケジューリングに採り入れることを提案する．命令列を遺伝的に取り扱うためにストリングやSMDといった新しい概念を導入する．これにより，GNUやSunのCコンパイラが提供するスケジューリングよりも高い性能を発揮でき，さらにマシン性能に依存しない安定した最適化が行えることを示す．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[†] and Yukio Umetani[††]

As RISC (Reduced Instruction Set Computer) processor becomes complex in recent years, the instruction scheduling plays a very important role as a mean for uncovering its potential performance. The application of genetic algorithm to the instruction sequence optimization is proposed and tested as a mean to squeeze out RISC processor performance. Also the new concepts “String” and “SMD” are introduced for that purpose. As a result, far better performances than those obtained by the GNU and Sun C compilers are obtained, and also the stability of performance is enhanced. The performance was 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 found that the conventional scheduling methods depend on a specific machine feature, and does not care for the data dependence via 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．これはどの時点をとらえてみてもプロセッサが複数の命令をパイプラインの異なるステージで実行しているためである．実行に余分な時間のかかるメモリ参照や浮動小数点演算や分岐命令は，通常のパイプライン処理を中断させるためにスループットを低下させる．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引き出すために命令をスケジューリングすることは非常に重要である．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，最近，急速にパイプライン制御方式を複雑化しているRISCプロセッサの性能を十分発揮していない恐れがある．

このような現状を打破するために筆者ら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[2)]．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

† 静岡大学大学院理工学研究科計算機工学専攻
Department of Computer Sciences, Graduate School 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

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[3]．

(1) 1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，目的関数の性質がよく分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うる可能性が高い．

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と考えた．

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ほかにレジスタ割当てやデータアドレス割当て，高度なコード最適化などの多くの要素が組み合わさっている．これらは RISC プロセッサの重要な性能要素であり，GA への応用も可能である．しかし，これらの問題は将来の課題として，本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，マシン特性に適応した安定した効果を期待できる．しかし，その一方で評価に多くの時間を費してしまうためにこのスケジューリングをあらゆる場面で適用するのは効率的ではない．したがって，実用面から考えた応用範囲としては，まず様々な機器に組み込まれていて半永久的にそのプログラムを使う組み込みシステム，特にリアルタイムでの性能が要求されるカーナビやエンジン制御などが考えられる．また，ハードウェア性能を評価するベンチマークテストにおいてはハードウェアの能力を極限まで引き出すソフトウェア技術として有効となる．さらに，フーリエ変換やガウス法などのライブラリ化された科学技術計算などの分野でその効果が期待できる．

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方式を示し，3 章でスケジューリング実験の結果を述べる．そして，4 章では遺伝的スケジューラと従来研究におけるスケジューリング方式との比較を行う．そして最後に，4 章で結論と今後の課題を述べる．

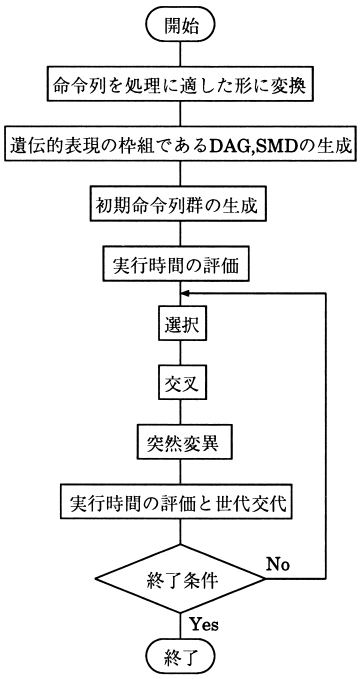

図 1 遺伝的スケジューラのアルゴリズム
Fig. 1 Genetic scheduling algorithm.

2. 遺伝的命令スケジューラ

2.1 アルゴリズム

我々は実際の遺伝的表現や遺伝的操作を考えるにあたっては，Breeder Genetic Algorithm（BGA）[4]の考え方を利用した．BGA は German National Research Center for Computer Science（GMD）が伝統的な家畜の品種改良で使われる方法をもとに開発した遺伝的アルゴリズムの方式である．BGA の主要な特徴は選択操作において，適合値に比例した選択に代えて，切捨て選択を行う点にある．切捨て選択は母集団のうちの最良の T %を選択してこれをもとに交叉や突然変異を行う方法である．この方法の長所は進化が評価関数と T 値選択に左右されず，また進化のペースダウンにも打ち勝てる点にあるとされている[4]．

図 1 に BGA 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示す．個々の遺伝的表現や遺伝的操作の詳細については後で述べる．手順としてはまずアセンブリー命令列[5),6)]をプログラム内での処理に適した形に変換して，これをもとに遺伝的表現の枠組みである DAG（Dependence Analysis Graph）と SMD（String Merge Diagram）を生成する．そしてこれをもとにあらかじめ定めた NPOPL 個の初期命

令列群を SMS（String Merge Status），SMO（String Merge Order）と呼ぶ遺伝的表現を使って生成する．これによって確定した各々の命令列を実際にマシン上で実行し，その実行時間を計測することによって評価を行う．この中から実行時間の短い順に NPOPL/2 個の命令候補群を取り出す選択を行う．そして選択した命令列群からランダムに NPOPL/2 個の対を作り，これらの対を交叉をさせることによって NPOPL 個の子孫を形成する．さらにこの NPOPL 個の交叉命令列群に対して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，新しい命令列に世代交代させる．世代交代では古い世代でも最小の実行時間を持つものだけは次世代に残し，それ以外はすべて新世代の個体に置き換える．この後に再びこの新世代の命令列に対し選択と遺伝的操作を行う．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を目的とする命令列とする．

2.2 遺伝的表現

我々は命令列の遺伝的表現の枠組みとして後に述べる SMD を用いるが，それは次に述べる DAG に基づいている．これらを順に説明する．

DAG（Dependence Analysis Graph）とストリング

DAG は命令間の依存による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．遺伝的スケジューラではこれらの依存関係を命令列から抽出して DAG を生成する．

例として，gcc でコンパイルした LFK（Livermore Fortran Kernels）11 の Sparc 用アセンブリー命令列の一部を取り上げる．**図 2** にアセンブリーコード，**図 3** に DAG を示す．図 3 において，アークが命令間の優先関係を示している．(1) はラベルであって実際には命令ではないが，スケジューラの内部では処理の関係で NOP 命令として扱っている．(1) はラベルなのでループ内の他のどの命令よりも先行しなければならず，これが制御依存である．また，(4)-(5) や (6)-(7) などは前者の命令で書き込んだレジスタ%f4 や%o3 を後者の命令で読み込んでいるので順序関係が定まり，データ依存制約となっている．(9) はループバック命令なのでループ内のどの命令よりも後に実行しなければならず，制御依存となっている．(10) は (9) のディレイスロットに位置する命令であり，(9) の実行の際に実行する命令で優先関係では (9) の前にくる．

```
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
Fig. 2 A part of Kernel11 assembly code.

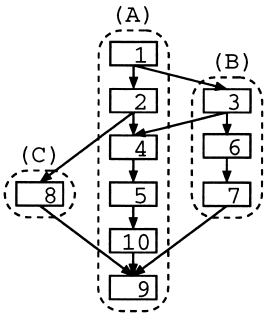

図 3 図 2 のコードの DAG
Fig. 3 DAG for Fig. 2.

このように順序の決まった命令列から依存関係のみ取り出した DAG を考えることによって，命令列に並べ換えの可能性を与えることができる．そしてこの命令列をうまくスケジューリングすると実行時間の改良が図れる．このスケジューリング処理を効率良く行うために DAG に対し次のような操作を行う．まず，DAG の先頭から依存アークを順にたどることによって部分命令列を切り出す．この部分命令列をストリングと定義する．この依存アークによる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り，命令列をストリングに分割した一例である．それぞれのストリングは命令相互の依存関係を保つかぎり並列実行が可能である．ストリングの生成には依存アークでの別れ道でどの命令をとるかによって，いくつかの可能性がある．どのような生成方法が命令スケジューリングに最良であるかはまだ研究中であるが，現時点では経験的に高い効果の得られたクリティカルパスを第 1 ストリングにする方法を採用している．クリティカルパスとは，DAG から生成しうる最長のストリングのことをいう．

SMD（String Merge Diagram）

GA で遺伝的操作を有効に機能させるためには，対象をコード化する遺伝的表現の定義方法が重要であるといわれている．先の節で定義した各々のストリングは命令間の依存関係を保つかぎり並列に実行可能であるが，これをうまく一本化（マージ）して命令列を生成すると，効率的に実行時間の改良が図れるということが命令列の解析により明らかになった．このため，筆者らはストリング間の一本化方法が命令スケジューリングの鍵を握ると考え，この一本化状態，すなわちあるストリングに属する命令を他のストリングのどの命令の後に組み込むかと，多数のストリングをどのような順番で一本化してゆくかを規定する遺伝的表現を考案した．

ストリングの一本化（マージ）が重要な理由は以下のとおりである．仮に単独のストリングをパイプライン計算機で実行する状況を想定すると，ストリングの定義から隣接する命令の間にはすべて依存関係があるのでパイプラインは空きが多いことが予想される．したがって，他のストリングの命令をその中にうまく配置すればパイプラインの空きを有効に利用できるであろう．前者を後者のマージ対象ストリングと呼ぶことにする．問題は，あるストリングを他のストリングにマージして全体命令列を組み上げる手順を決めることである．マージはどのストリングの間でも行うことができるが，命令間の依存制約を守る必要があるので，それを早期に守るために直接のマージ対象を，相手方の命令との間に依存制約を持つストリングに限定することとする．この方法でストリングを順に取り上げ，未処理のストリング群の中からマージ対象を選択してゆくとストリング全体をマージの階層グラフに組み上げることができる．このグラフを SMD（String Merge Diagram）と名付ける．SMD はサイクルを持たず最上位のストリングを基点としてレベル付けを行うことができる．SMD は全体命令列の組み立てをマージの階層に分解して示す枠組みであり，各ストリングの親ストリング（マージ対象ストリング）へのマージ方法と同レベルのストリングのマージ順を決めれば全体命令列が決定される．前者を SMS（String Merge Status），後者を SMO（String Merge Order）というコードで表現することとする．SMS は各ストリングに対応する独立した複数の部分コードの集まりであり，SMO も各レベルごとに独立したマージ順の集まりである．このようなコード構成は特に GA を特徴付ける交叉の実現に有利と考えた．交叉は部分コードの交換により，独立した，あるいは相乗作用のある良い因子群を 1 つの個体に集結させる効果を狙っている．たとえば，SMS においては親ストリングを共有する同レベルのストリングが複数あるような状況でそのような効果が期待できる．

SMD の構成手順は以下のとおりである．まず，ストリングに次の方法でレベル付けを行う．そのためにストリングの間の依存関係を定義する．ストリング (A) を構成する命令とストリング (B) を構成する命令の間に 1 つ以上の依存関係を持つとき，(A) と (B) は相互に依存すると定義する．この依存関係は命令間の場合と異なって方向性がなく反射的である．はじめに全ストリングのレベルをゼロに初期化する．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．それ以降レベル $i\,(i = 0, 1, 2, \cdots, n)$ 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 $(i+1)$ とする．これをすべてのストリングがレベル付けされるまで繰り返す．図 3 の例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．このレベルをストリングをマージしていく順序を決定する指針とする．

このレベル分けしたストリング群に基づいて SMD を構成する．SMD はストリングに SMS（String Merge Status）と SMO（String Merge Order）という表現を加えて構成する．SMS は 1 より高いレベルのストリングに属する各命令を，それより低いレベルのストリングに属するどの命令の後に挿入すべきかを示すコードである．SMO は挿入すべき位置の定まった命令を持つ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．最初の SMD の生成では SMO と SMS として，当初の命令列順序とできるだけ同じように生成できるものを指定する．たとえば，命令 i の SMS コードは，i より小さい番号のうちで，とりうる命令の最大番号の命令とする．

図 4 に図 3 に対する SMD の一例を示す．図 4 ではレベル付けされた (A)，(B)，(C) のストリングが存在し，レベル 2 以上のストリングの各命令はそれぞれ下位レベルのどの命令の後に挿入するかを示す SMS コードを有している．たとえば，ストリング (B) の命令 (3)(6)(7) の SMS コードがそれぞれ (2)(5)(5) と

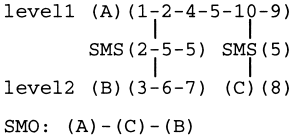

図 4 図 3 の SMD
Fig. 4 SMD for Fig. 3.

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．ストリング (C) についても同様である．同じレベルのストリングが複数存在するときなどには，ストリングを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) の順序になっており，ストリング (C) を (A) にマージした後に (B) をマージすることを指定している．この SMS と SMO により，図 4 の SMD で生成される命令列は (1)(2)(3)(4)(5)(6)(7)(8)(10)(9) となる．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（new）

SMS と SMO を DAG の制約内で乱数により決定して命令列を生成する．生成に失敗したら，再試行し成功するまで繰り返す．詳しいアルゴリズムは**図 5** のようになっている．

例として，図 5 のアルゴリズムの流れを**図 6** の SMD を使って説明する．手順 (1) の SMO の生成ではレベル 2 について示す．手順 (1-1) で STRQ に (B)-(C)-(D) をセットする．そして，手順 (1-2) により乱数を利用した交換が行われる．これによって 1 番目の要素が 3 番目の要素と交換され，2 番目の要素が自分自身と交換されたとすると，STRQ は (D)-(C)-(B) となる．この STRQ が手順 (1-3) で SMO にセットされ，図 6 の SMO の一部を形成する．他のレベルも同様である．手順 (2-1) ではレベル 1 のストリングである (A) の SMS コードを 0 にセットして，図 6 の SMS の一部を形成する．手順 (2-2) では (E) の SMS コードの生成を示す．手順 (a) の (i) でまず，図 6 で (E) が依存するストリング (A) と (B) が STRQ にセットされ，手順 (ii) のソートにより (A)-(B) となる．そして，手順 (iii) により，ISQ は 1-2-3-4-5-6-7-8-9-11-12-18-17 となる．手順 (b) では (i) で図 6 より命令 7 から 18 までが挿入できる範囲であると分かり，ISQ と照らし合わせて *cllim* と *chlim* にはそれぞれ 7 と 12 が入る．手順 (ii) では ($7 < 12$) で手順 (iii) に進み，7 から 12 の間で乱数を発生させる．これによって 11 が選択されたとするとこれに対応する ISQ の命令 12 が (E) の SMS コードとなる．他のストリングの SMS コードも同様に決定する．

(2) 選択（selection）

選択は各個体の評価値によって，次世代に残す個体群を確率的に決定する．

```
データ構造
 STRQ: string番号を格納する１次元配列で格納string
       数を保持する領域も持つ．
 ISQ: 命令番号を格納する１次元配列で格納命令数を保持
      する領域も持つ．
 chlim, cllim: 命令番号をISQ内の位置により指定し，
       それぞれ選択可能な命令の上限と下限を示す．

アルゴリズム
(1)SMOを生成する．レベルごとに昇順に次に示すnewSMO
   関数により決定する．
 ・newSMO関数(引数 レベル番号，SMO)
 レベル番号の示すSMOコードを生成してSMOにセットする．
 (1-1)指定レベルの全string番号をSTRQ配列にセットす
      る．
 (1-2)１番目のSTRQ要素(string番号)を，STRQの全要
      素から乱数発生関数により選択したものと入れ換え
      る．この操作を２番目要素以降繰り返す．
 (1-3)この結果できたSTRQをSMOにセットする．

(2)SMSを生成する．
 (2-1)レベル１に属する命令のSMSコードを全て０にセッ
      トする(レベル１の命令はマージしないため)．

 (2-2)レベル２からはレベルごとに昇順に，さらにレベ
      ル内で(1)で決定したSMOの順にstringごとに，次
      に示すnewSMS関数でSMSを決定する．
  ・newSMS関数(引数 string番号，SMS，SMO)
  指定したstring番号に属する命令のSMSコードを決定し
  てSMSにセットする．
  (a)引数のstringと依存関係を持ち，さらに自分より下
     位レベルのstringで構成する部分命令列を次に示す
     getDISQ関数により生成し，整数型のISQ配列にそ
     の命令順序をセットする．SMSコードはこれらの命令
     から選ばれる．
   ・getDISQ関数(引数 string番号，ISQ，SMS，SMO)
   ISQに，引数のstringのSMSコード候補(すなわち，依
   存している命令)となる部分命令列をセットする．
   (  i)SMD情報から，引数のstringが依存するstring
        を求めて，STRQに格納する．
   ( ii)このSTRQをSMOにしたがってソートする．
   (iii)STRQのstringを生成済みのそれらのSMOとSMS
        にしたがって，マージして部分命令列を生成し，
        これをISQにセットする．

  (b)string内の各命令のSMSコードを，依存の優先順序
     にしたがって，ISQの命令内から次のように決定す
     る．
   (  i)DAG情報から，SMSコードを決めようとしている
        命令が，ISQ内のどの範囲に挿入できるかを求め
        て，その範囲のISQの最小と最大の配列添字をそ
        れぞれ整数型のcllimとchlimにセットする．
   ( ii)(cllim>chlim)ならば(stringとレベルの概念
        の導入により起こり得る)，挿入位置が存在しない
        ので，命令列生成失敗として終了する．
   (iii)(cllim<=SMSコード<=chlim)の範囲で乱数発
        生関数によりSMSコードを決定する．
```

図 5 初期命令群生成アルゴリズム
Fig. 5 Algorithm for new population generation.

ここでは GMD の BGA にしたがって母集団のうちの上位の T %以外の切捨て選択を行う．アルゴリズムとしては，NPOPL 個の命令列から実行時間の評価値により最良の NPOPL/2 個を選択して遺伝的操作の対象命令列とする．したがって，$T = 50$ である．

(3) 交叉（crossover）

交叉は 2 つの命令列間で遺伝的表現の一部を交換することによって新しい個体を生成するものである．ここで両親の優れた部分形質をうまく組み合わせて，子に継承させることに成功すれば，探索における飛躍をもたらす．

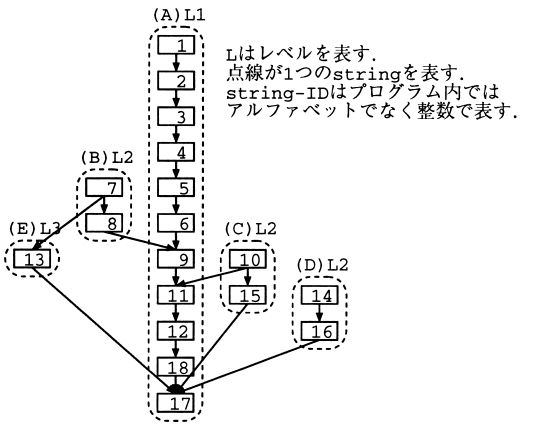

図 6 ある命令列の SMD
Fig. 6 An example of SMD.

実際には，SMS においてはストリングを単位として，SMO においては特定レベルのストリング群を単位としてそれらの一部を対の命令個体の間で交換する．交換に際してはそれぞれ条件があり，SMS では交換の対象とするストリングの SMS コードが両方で異なり，さらに対象とするストリングと依存関係を持つ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 では選択したレベルの SMO が両方で異なることである．この条件の下にある規定回数だけ成功を目指してトライする．規定回数実行しても失敗した場合は親の性質をそのまま子に継承する．詳しいアルゴリズムは**図 7** のようになっている．

図 7 のアルゴリズムの流れを，図 6 と**図 8** の SMD を使って説明する．まず，手順 (1) で選択された命令列のうちの 1 組が図 6 と図 8 であるとする．手順 (2) ではストリング (B)，(C)，(D)，(E) について条件を満たすか調べてみる．まず，(B) について手順 (2-1) での条件は互いに 6-6 と 4-7 であるので，異なっていて満たす．手順 (2-2) では図 6 よりストリング (B) と (E) が STRQA に集められる．これは交叉対象のストリングが (B) となった場合には，これに依存する (E) の SMS もともに交換するためである．手順 (2-3) では STRQA が依存するストリングは (A) であり，このストリングで構成する命令列は図 6 と図 8 で同じなので，条件を満たし候補となる．(C)，(D) は手順 (2-1) にて候補から外される．ストリング (E) についてはまず，手順 (2-1) の条件は満たすことは明

データ構造
STRQS, STRQO, STRQA: STRQと同じ.

アルゴリズム
(1) 交叉のペアを選択操作によって対象となったNPOPL/2の命令列から乱数発生関数によりNPOPL/2組生成する．それぞれのペアについて次からの操作をする．

(2) レベル 2 以上のstringについて(SMSコードを持つから)，次の方法でSMS交叉の候補となるstringを整数型のSTRQS配列に集める．
(2-1) お互いのSMSコードが異なっているかを調べる．同じならば，交換が無意味となるので候補としない．
(2-2) 自分と自分に依存する上位レベルのstringをSTRQAに集める．
(2-3) getDISQ関数によりこのSTRQAが依存する下位レベルのstringで構成する命令列を求めて，この命令列がペアの両者で同じかを調べ，同じならば，候補としてSTRQSに加える．

(3) SMO交叉の候補stringを整数型のSTRQO配列に集める．レベル 2 以上のstringについて，stringごとに自分が属するレベルのSMOがお互いに異なっているか調べ，異なっていれば候補としてSTRQOに加える．

(4) SMSとSMOのどちらを交叉させるかを次のように決める．
(4-1) 要素数STRQS=STRQO=0ならば，候補なしで終了．
(4-2) 一方のみが0ならば，0でない方を交叉させる．候補のSTRQから乱数発生関数により交叉の対象とするstringを決定する．
(4-3) STRQS, STRQOともに存在するならば，(1/全候補)の確率でSMOを，それ以外ではSMSの交叉を選択する．選択したSTRQから乱数発生関数により対象stringを決定する．

(5) 交叉を実行する．

SMSの場合，SMD情報により，選択したstringとこれに依存するstringのSMSをお互いに交換する．

SMOの場合，選択したstringの属するlevelのSMOをお互いに交換する．

図 7 交叉のアルゴリズム
Fig. 7 Crossover algorithm.

```
SMO: (A)-(D)-(C)-(B)-(E)
SMS: (A) 0-0-0-0-0-0-0-0-0-0-0
     (B) 4-7
     (C) 9-12
     (D) 12-12
     (E) 11
```

図 8 対となる命令列の SMD
Fig. 8 SMD for another instruction sequence.

らかである．手順 (2-2) では STRQA は (E) となり，手順 (C) では STRQA が依存するストリングは (A) と (B) であり，これらが構成する命令列は，図 6 では 1-2-3-4-5-6-7-8-9-11-12-18-17，図 8 では 1-2-3-4-7-8-5-6-9-11-12-18-17 となり，これは互いに異なっているので，候補とはならない．その結果，STRQS は (B) のみを含む．手順 (3) では互いに SMO が同じであるために候補となるストリングは存在せず，STRQO は空となる．手順 (4-1) では STRQS に要素が存在するので，(4-2) に進んで STRQS が選択される．ストリング (B) が選択されたとすると，手順 (5) で (B) と (E) の SMS が互いに交換される．

次に交叉がもたらす命令列の実際の変化を図 6 と次に示す**図 9** の SMD を使って示す．まず，それぞれの

```
SMO: (A)-(D)-(B)-(C)-(E)
SMS: (A) 0-0-0-0-0-0-0-0-0-0-0
     (B) 4-7
     (C) 9-12
     (D) 4-12
     (E) 12
```

図 9 対となる命令列の SMD
Fig. 9 SMD for another instruction sequence.

SMD が生成する命令列は

- (1)(2)(3)(4)(5)(6)(7)(8)(9)(10)(11)(12)(13)(15)(14)(16)(18)(17)
- (1)(2)(3)(4)(14)(7)(8)(5)(6)(9)(10)(11)(12)(13)(16)(15)(18)(17)

である．ここでストリング (B) の SMS，すなわち (6-6) と (4-7) を互いに交換することを考える．するとそれぞれの命令列は，

- (1)(2)(3)(4)**(7)(8)**(5)(6)(9)(10)(11)(12)(13)(15)(14)(16)(18)(17)
- (1)(2)(3)(4)(14)(5)(6)**(7)(8)**(9)(10)(11)(12)(13)(16)(15)(18)(17)

となり，太字の命令の位置がそれぞれ変化する．また，SMO でレベル 2 のストリング，すなわち (D)(C)(B) と (C)(B)(D) を交換すると

- (1)(2)(3)(4)(5)(6)(7)(8)(9)(10)(11)(12)(13)**(14)(16)(15)**(18)(17)
- (1)(2)(3)(4)**(7)(8)(14)**(5)(6)(9)(10)(11)(12)(13)**(15)(16)** (18)(17)

となり，レベル 2 に属する命令の位置が変化する．

(4) 突然変異（mutation）

突然変異は命令列のある遺伝子を別の遺伝子に置き換えることにより，個体の近傍に新しい個体を生成するもので，その役割としては，交叉で得られた解の近傍の探索と，交叉だけでは得られないパターンの生成という 2 つがある．これにより局所解からの脱出が可能となる．

実際には，レベル 2 以上のストリングの中から確率的に選択した命令の SMS コードを変化させ，さらに選択したレベルの SMO を確率的に変化させる．アルゴリズムは，**図 10** のようになっている．

図 6 を使って図 10 のアルゴリズムの流れを示す．まず，手順 (1) では上限回数は $(5*10)$ となる．手順 (2) では変異確率は $(1/(7+3))$ となる．手順 (3) ではストリング (C) を例に使って示す．まず，手順 (3-1) で図 6 よりストリング (C) の ISQ は 1-2-3-4-5-6-9-11-12-18-17（ストリング (A) そのもの）となる．手順 (3-2) では初めに先頭の命令 10 を選択する．手順 (a) では SMS の範囲は図 6 より 0 から 9 までであるので，ISQ と照らし合わせて *cllim* が 0 で *chlim* が 7 となる．手順 (b) は $(0 < 7)$ でパスする．手順 (c) は命令 10 の SMS は 9 であるので，*cidx* は 7 となる．手順 (d) で変異確率がヒットしたとすると，手順 (i) では $range = \max(7-7, 7-0) = 7$ となる．手順 (ii) では 111 となる．手順 (iii) では *disp* が $011 = 3$ となったとする．手順 (iv) は $(3 < 7)$ となり，手順 (v) で負が選択されたとすると，手順 (vi) では $(0 < j = 7-3 < 7)$ となる．手順 (vii) はスキップし，手順 (viii) で $j = 4$ に対応する ISQ の 4 が新しい SMS コードとなる．手順 (4) は初期命令の生成と重複するので省略する．

```
データ構造
 ISQ: 部分命令列と命令数を格納する 1 次元配列(図 5 に
      同じ).
 cidx: 突然変異の対象とするSMSコードの位置をISQの
       配列添字として保持する.
 range, disp, j: 整数型の変数.
 chlim, cllim: 図 5 に同じ.

アルゴリズム
(1)突然変異演算の試行上限回数をstring数*10にセット
   する.

(2) 1 回あたりの変異確率を(1/(レベル 2 以上のSMSコー
   ド数+全level数(突然変異可能な全対象数)))にセット
   する.

(3)SMSの突然変異をさせる．レベル 2 からレベルごとに昇
   順に，レベル内ではSMOの順序でstringごとに，次に
   示すmodSMS関数により指定したstringのSMSを変化さ
   せる.
 ・modSMS関数(引数 string番号，SMS，SMO，変異確率)
 (3-1)getDISQ関数で当該stringをマージする相手とな
      る命令列を求め，ISQにセットする.
 (3-2)引数のstringに属する命令のSMSコードを初めから順
      に変位させる.
  (a)変位可能な上限と下限となるchlimとcllimを計算
     する.
  (b)chlim<cllimならば，変位できないので，突然変異
     失敗でmodSMS終了.
  (c)整数型のcidxに現在のSMSコードのISQ上のインデッ
     クスを代入する.
  (d)変異確率がヒットしたら，cidxの値に次の方法で決
     定する整数dispを加える.
   (  i)range=max(chlim-cidx,cidx-cllim)とする
      .
   ( ii)rangeを 2 進数に変換する.
   (iii)下位の桁から順に(1/(2**(桁)))の確率で1を
        たて，この結果をdispにセットする.
   ( iv)(disp>range)ならば，dispの最上桁を無効と
        する.
   (  v)1/2の確率でdispの正負の変異方向を決める.
   ( vi)j=cidx+dispとしてcllim<j<chlimを満たすか
        チェックする.
   (vii)満たさないならば，j=cidxとしてdisp=0とする
      . したがって変異量0も突然変異したものとみなす.
   (viii) ISQ.isn[j]を新しいSMSコードとする.

(4)SMSの突然変異に成功したら，引続きSMOの突然変異を
   実行する．レベルごとに昇順に変異確率にしたがって
   newSMO関数によりSMOを再設定する.
```

図 10 突然変異のアルゴリズム
Fig. 10 Mutation algorithm.

次に突然変異がもたらす命令列の変化を図 6 を使って示す．図 6 においてストリング (C) の (10) の SMS コードが (9) から (4) に変異したとすると命令列は

- (1)(2)(3)(4)**(10)**(5)(6)(7)(8)(9)(11)(12)(13)(15)(14)(16)(18)(17)

となる．また，SMO のレベル 2 の部分が (D)(C)(B) から (B)(D)(C) に変異すると

- (1)(2)(3)(4)(5)(6)(7)(8)(9)(10)(11)(12)(13)**(14)(16)(15)**(18)(17)

となる．

交叉と突然変異の間には，相補的な側面と競合的な側面が存在する[3]．集団全体が探索空間のある超平面に陥ってしまい交叉だけでは抜け出せない場合，突然変異により脱却が可能となることが期待できること，および一般に大きな変異をもたらす交叉に対して突然変異による局所的な探索が有効に働くこともありうるという点が，相補的である．しかし，交叉によって形成された優れた形質が突然変異により破壊されてしまうことがある点では，競合的である．

3. LFK による性能評価

この章では LFK ベンチマーク[7]を利用して GNU-C コンパイラ gcc-2.8.1 と Sun WorkShop Compiler cc-4.2 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．gcc によるアセンブリーコードを対象とした実験では 3 つの UNIX マシン Ultra2（333 MHz），Ultra1（143 MHz），SPARC（100 MHz）で実験を行い，マシン特性による性能の変化も調べる．cc によるコードでは今回は Ultra2 のみで行う．

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性能の解析を行いやすくするために 1 つのプログラムを 24 個の独立プログラムに分割し，さらに C コンパイラでのスケジューリングと比較するために言語を FORTRAN 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には UNIX の組み込み関数である clock を使用した．clock はオンライン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，10 ms が分解能の限界であった．

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調整した．Ultra1，2 と SPARC ではマシン性能が相当に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) カーネルを gcc では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 Tiemann[8]によるスケジューリングが行われる．cc では標準的な最適化オプションである xO2 と，オプティマイザ用のプロセッサを指定するオプション xchip を ultra にしてアセンブリコードを生成する．マニュアルによると xchip オプションは指定したプロセッサのタイミング特性を使用して命令スケジューリングに影響を与える．

(2) 遺伝的スケジューリングはこのアセンブリーコードのループを中心とした部分を対象として行う．機種依存性を減らすために，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 プロセッサ命令列に変換する．

(3) アセンブリー命令列ファイル，スケジュール対象部分の仮想 RISC 命令列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，進化の過程と準最適な命令列を得る．

(4) この操作を 24 個のカーネルに対して，gcc では 3 つのマシンで cc では 1 つのマシンで行う．

図 11 に LFK7 を対象とした遺伝的スケジューラの出力の一部を載せる．これは 1 世代分の進化過程を表している．2 行目は，初期命令群の数 NPOPL が 10 であることを示している．3 行目は，最初の世代の母集団の命令列の最小，最大，平均の実行時間を示している．4 行目は，切捨て値 50%のもとに遺伝的操作の対象として選択された最良の 5 命令列の ID を表している．5 行目から 14 行目までは交叉の過程を表している．5，6 行目を説明すると，4 行目で選択された 5 命令列のうち交叉の対象として 0 番目の ID1 と 2 番目の ID6 の命令列を選択し，これらの命令列のストリング ID が 11 のものの SMS コードを交換している（choice の値が 1 のときは SMS を 2 のときは SMO を交換するようになっている）．15 行目から 40 行目までは突然変異の過程を示している．交叉の結果，生成された 10 命令列を順番に突然変異させている．15 行目は，0 番目の命令列を突然変異させたが，変化しなかったことを示している．21，22 行目は 4 番目の命令列を突然変異させて，命令 ID62 の SMS コードが実際に変化していることを示している．25 行目では 5 番目の命令列の突然変異の一部が失敗していることを示している．そして最後に 41 行目でこの遺伝的操

```
  breeder algorithm test                          (1)

NPOPL 10                                          (2)
gs 0 min 1250000 max 1750000 avr 1495000          (3)
 1 3 6 7 8                                        (4)

 k 0 1 2                                          (5)
 mate strnum 11 choice 1                          (6)

 k 2 1 4                                          (7)
 mate strnum 10 choice 1                          (8)

 k 3 1 0                                          (9)
 mate strnum 2 choice 1                           (10)

 k 3 1 0                                          (11)
 mate strnum 3 choice 1                           (12)

 k 0 1 1                                          (13)
 mate strnum 7 choice 1                           (14)

 mutate k 0                                       (15)

 mutate k 1                                       (16)

 mutate k 2                                       (17)
 mutate isn 4 choice 1                            (18)
 mutate isn 7 choice 1                            (19)

 mutate k 3                                       (20)

 mutate k 4                                       (21)
 mutate isn 62 choice 1                           (22)

 mutate k 5                                       (23)
 mutate isn 57 choice 1                           (24)

 SMS modification failed. modSMS 1                (25)

 mutate k 6                                       (26)
 mutate isn 6 choice 1                            (27)
 mutate isn 44 choice 1                           (28)
 mutate isn 53 choice 1                           (29)

 mutate k 7                                       (30)

 mutate k 8                                       (31)
 mutate isn 31 choice 1                           (32)
 mutate isn 3 choice 1                            (33)
 mutate isn 6 choice 1                            (34)

 mutate k 9                                       (39)
 mutate isn 59 choice 1                           (40)

gs 1 min 1230000 max 1790000 avr 1419000          (41)
...
```

図 11 LFK7 の遺伝的進化過程の一部
Fig. 11 A part of LFK7's genetic evolution process.

作の結果を 3 行目と同様な形式で示している．3 行目と比較してみると改良の様子が分かる．

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 ms 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．ここで準最適命令列は機種ごとにすべて異なるものである．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載せている．

表 2 には Sun-C コンパイラと遺伝的スケジューラのコードによる Ultra2 上の実行時間の結果を載せる．Ultra1，SPARC での結果は同様な傾向であるので省略した．表の構成は対象マシン数が 1 つであるという点を除けば，表 1 と同じである．

図 12 には gcc における LFK7 についての遺伝的スケジューラによる実行時間の進化の様子を示してある．縦軸が実行時間を表し，横軸が遺伝的スケジューラ内での世代を表す．1538 ns のところの点線が GNU-C コンパイラが生成したコードによる実行時間を示している．そして，灰丸，白丸，黒丸はそれぞれ各世代での最大，最小，平均の実行時間を示している．この図 12 より，最小実行時間が世代を重ねるにつれて向上していく様子が分かる．さらに最初の世代の初期命令群を評価した時点で，コンパイラのスケジューリングよりも良い実行時間を得ることに成功している．

遺伝的スケジューラの実行時間は，スケジューラが評価として命令列を実際に実行するためスケジュール対象とする命令列の実行時間に依存する．Ultra2 上の gcc コードでは，およそで LFK3 では 3 分 15 秒，LFK7 では 3 分 20 秒，LFK18 で 2 分 45 秒，LFK11 で 2 分 30 秒，LFK23 で 4 分の時間を要する．また，初期命令列群数と世代交代数によって評価のために実際に命令列を実行する回数が変化するので，これらにも依存する．

3.3 考　　察

表 1 の実験結果より Ultra2，1 ではすべ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性能の悪化した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，すべてのケースで 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 8) 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 8) 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

表 1 機種による改良度の比較（GNU-C コンパイラ）
Table 1 Performance table for three machines (GNU-C compiler).

NAME	Ultra2			Ultra1			SPARC			IS-
	Copt-Exc (ms)	Sopt-Exc (ms)	S/C* 100 (%)	Copt-Exc (ms)	Sopt-Exc (ms)	S/C* 100 (%)	Copt-Exc (ms)	Sopt-Exc (ms)	S/C* 100 (%)	Amo
kernel1	1632.5	1211.5	74.2	3405.0	2561.5	75.2	2970.0	2973.5	100.1	49
kernel2	1073.5	867.0	80.8	2252.0	1802.0	80.0	2740.0	2691.0	98.2	54
kernel3	1524.5	1478.5	97.0	3177.0	3066.0	96.5	2096.0	1995.0	95.2	58
kernel4	1473.5	1337.5	90.8	2942.0	2652.0	90.1	3098.5	3098.0	100.0	42
kernel5	1594.0	1381.0	86.6	2991.5	2571.0	85.9	2040.5	1940.0	94.9	33
kernel6	1102.5	935.0	94.8	2045.5	1757.5	85.9	2111.3	1945.0	92.1	48
kernel7	1538.0	1196.5	77.8	3022.5	2286.5	75.6	3325.5	3113.5	93.6	77
kernel8	968.5	874.0	90.2	1764.0	1637.0	92.8	1875.5	1838.0	98.0	177
kernel9	1543.0	1286.5	83.4	2939.5	2415.5	82.2	2881.5	2983.0	103.5	79
kernel10	1731.0	1692.5	97.8	3080.5	3024.5	98.2	4760.5	4711.5	99.0	52
kernel11	1308.5	1105.5	84.5	2755.5	2339.5	84.9	1570.0	1572.5	100.2	30
kernel12	1171.5	1005.0	85.8	2479.0	2124.5	85.7	2788.0	2784.0	99.9	28
kernel13	1044.0	960.0	92.0	1968.0	1741.5	88.5	1221.0	1207.5	98.9	103
kernel14	1229.0	1074.0	87.4	2131.0	1942.5	91.2	1431.5	1408.5	98.4	149
kernel15	1337.0	1294.0	96.8	2716.0	2614.0	96.2	1941.5	1938.0	99.8	228
kernel16	866.0	801.0	92.5	1808.5	1674.0	92.6	1546.5	1532.0	99.1	135
kernel17	1320.5	1210.0	91.6	2758.0	2555.5	92.7	2598.5	2521.0	97.0	77
kernel18	1232.5	1160.0	94.1	2306.0	2188.0	94.9	1855.5	1949.0	105.0	242
kernel19	1176.5	1071.0	91.0	2473.0	2251.0	91.0	2073.5	2017.0	97.3	46
kernel20	1583.0	1468.0	92.7	2986.5	2921.0	97.8	1852.5	1858.5	100.3	102
kernel21	1494.0	1428.0	95.6	2615.0	2535.5	97.0	3117.0	3111.0	99.8	45
kernel22	1439.5	1437.0	99.8	3007.5	3004.5	99.9	2173.0	2159.0	99.4	55
kernel23	2160.0	1760.0	81.5	4123.5	3054.5	74.1	3119.0	3180.0	102.0	80
kernel24	1070.0	1066.0	99.6	2426.5	2419.0	99.7	1910.0	1908.0	100.0	34

表 2 遺伝的スケジューラによる ultra2 での LFK の実行時間（Sun-C コンパイラ）
Table 2 Performance table (Ultra2, Sun-C compiler).

NAME	Copt-Exc (ms)	Sopt-Exc (ms)	S/C* 100 (%)	IS-Amo
kernel1	1075.0	1078.0	100.3	51
kernel2	1297.5	1241.0	95.6	81
kernel3	406.0	404.5	99.6	34
kernel4	1368.5	1379.5	100.8	62
kernel5	1710.5	1666.5	98.2	45
kernel6	1233.5	1234.5	100.8	65
kernel7	1973.5	2171.5	110.0	106
kernel8	1289.0	1298.5	100.7	196
kernel9	1736.5	1611.5	105.3	74
kernel10	2238.0	2240.0	92.8	74
kernel11	1408.5	1411.5	100.2	45
kernel12	1174.0	1175.0	100.1	37
kernel13	1363.5	1239.0	90.9	162
kernel14	1479.5	1539.5	104.0	228
kernel15	2234.0	2168.5	97.1	304
kernel16	1198.0	1200.5	100.2	179
kernel17	1769.5	1922.0	108.6	119
kernel18	1493.0	1474.0	98.7	402
kernel19	2347.5	2346.5	99.9	77
kernel20	1863.0	1932.5	103.7	137
kernel21	1902.5	1909.0	100.3	62
kernel22	1452.5	1449.5	99.8	71
kernel23	2515.0	2363.5	94.0	116
kernel24	1311.0	1302.5	99.4	44

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．先に載せた図 2 は Kernel11 の核となるループ部分であり，**図 13** はその DAG で隣接する命令間のデータ依存を太線で示している．ここで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 Kernel11

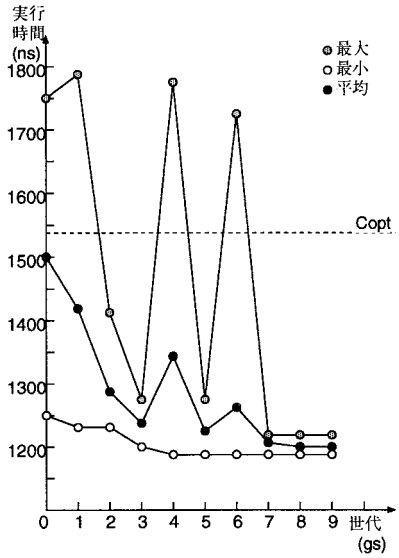

図 12 gcc における LFK7 の進化課程
Fig. 12 Evolution process of LFK7 (gcc compiler).

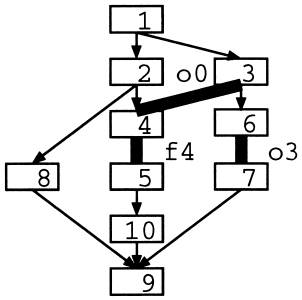

図 13 Kernel11 の DAG
Fig. 13 DAG of kernel11.

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たくさん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 3** にその結果を載せる．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変えると，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いえる．

表 2 の実験結果より，Sun-C コンパイラのスケジューラと遺伝的スケジューラでは実行時間は同程度であることが分かる．これは，Sun のコンパイラが xchip オプションによりプロセッサのタイミング特性の情報をパラメータとして採り入れた高度な最適化，スケジューリングを実行して，最適に近くスケジューリングした命令列を生成しているためである．

表 3 機種による遺伝的コードの違い
Table 3 Difference of performance between machines.

NAME	Ultra2		Ultra1	
	U2-code	U1-code	U1-code	U2-code
kernel7	1196.5	1206.0	2286.5	2416.5
kernel9	1286.5	1309.5	2415.5	2509.0
kernel23	1760.0	1795.5	3054.5	3097.5

しかしその一方で，それぞれのコンパイラによるコードの実行時間を比較してみると，商用コンパイラである Sun のコードの方が大部分で実行時間が悪いのが分かる．これは Sun のコンパイラがスケジューリング以外の最適化処理での改良の余地があることを示していると思われるが詳細はさらに調査を要する．

4. 従来研究との比較

命令スケジューリングはプログラムの意味を変えないで，その実行時間（インタロックの発生）をできるだけ少なくするように，半順序集合である DAG をトポロジカルソートすることである．

これまで，命令スケジューリングに関した研究は分岐スケジューリング，リストスケジューリング，トレーススケジューリング，ソフトウェアパイプライニングなどの手法[9]が考案され，スケジューリングコストとスケジューリングによって得られたコードの性能のバランスを考えて様々な手法を組み合わせて使ってきた．分岐スケジューリングは遅延分岐を用いたアーキテクチャで適用され，これは分岐命令の後の遅延スロットに nop 以外の有用な命令を置くものである．遺伝的スケジューラは分岐スケジューリングの機能を含んでいる．リストスケジューリングは局所的な基本ブロック内でのスケジューリングでまず DAG と各命令のスケジューリング優先度を求めて，この DAG を基にその時点で実行可能な命令のリストを作り，このリストから優先度に基づいて命令をスケジューリングするものである．各命令の優先度はその命令とその命令の次に続くことのできる命令の間における遅延に基づいていてその値はアーキテクチャごとに固有であり，この遅延の大きさに比例して優先度が高くなる．遺伝的スケジューラでは DAG に基づいて命令を構成するために遅延分岐やリストに基づくスケジューリングが偶然的な要素で行われる．実際に今回の実験で比較対象とした GNU コンパイラのスケジューラはリストスケジューリングと分岐スケジューリングを組み合わせて使っていて，理論的には効率的にスケジューリングさ

れるはずだが，遺伝的スケジューラと比較するとこれまでに示したような差が出た．これは優先度の決定方法に改良すべき点があることを示している．トレーススケジューリングは大域的なスケジューリングで基本ブロック単位の PDG（プログラム依存グラフ）を作成し，ブロック間で命令を有用にもしくは投機的に移動するものである．現在の遺伝的スケジューラでは基本ブロックをまたぐスケジューリングは行っていない．ソフトウェアパイプライニングはループに対して大きな効果を発揮するもので，ループの複数回の繰返しを対象に変数を拡張したりレジスタをリネームしたりして命令レベルの並列化を引き出すものである．現状の遺伝的スケジューラではループ構造を意識しないが，ループ展開されたコードを対象にスケジューリングすれば同等の結果を期待できる．

これらのスケジューラを実現するにはアーキテクチャの詳細を知る必要があり，結果としてできたスケジューラは前提とするアーキテクチャ特性にそぐわなければ性能が発揮できなくなる．

遺伝的命令スケジューリングでは確率的，偶然的要素によっているためアーキテクチャ特性に依存しにくく，また実行時間をそのまま評価に使うだけなので詳細なアーキテクチャ特性を知る必要がないために適用が比較的簡単である．

5. 結　　論

今回の研究では遺伝的スケジューラを GNU と Sun の 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，遺伝的スケジューラで発見された命令列をヒントにして，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．さらには遺伝的スケジューラをパソコンの CPU（Pentium II など）を対象にして評価し，またキャッシュの利用効率向上を目的とした適用実験を行う．また，ストリングの切り出し方法には自由度がありその抽出が GA の効率を左右すると予想される．この点も今後追求してゆきたい．

謝辞　本研究は平成 10 年度，11 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) Umetani, Y.: Application of genetic algorithm to instrucion 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，朝倉書店 (1996).
3) 北野宏明：遺伝的アルゴリズム，産業図書 (1993).
4) Muehlenbein, H. and S-Voosen, D.: Predictive models for the breeder genetic algorithm i, *Evolutionary Computation 1*, MIT Press (1993).
5) Inc. SPARC International: *The SPARC Architecture Manual Version 8*, Prentice-Hall (1992).
6) Sun Microsystems: *Sun-4 Assembly Language Reference Manual* (*Revision A of 27*), Sun Microsystems (1990).
7) McMahon, F.H.: The livermore fortran kernels: A computer test of numerical performance range, Technical Report, Lawrence Livermore National Laboratory (Dec. 1986).
8) Tiemann, M.: The gnu instrucion scheduler, Technical Report, Free Software Foundation, Cambridge, MA (June 1989).
9) Muchnick, S.S.: *Advanced COMPILER DESIGN IMPLEMENTATION*, Morgan Kaufmann (1997).

(平成 11 年 6 月 8 日受付)
(平成 12 年 1 月 6 日採録)

伊東　勇（学生会員）

昭和 50 年生．平成 10 年静岡大学工学部情報知識工学科卒業．同年静岡大学大学院理工学研究科計算機工学専攻入学．現在，在学中．遺伝的アルゴリズムを用いた命令スケジューリングの研究に従事．

梅谷 征雄（正会員）

昭和 19 年生．昭和 43 年東京大学理学部数学科卒業．同年,（株）日立製作所に入社．中央研究所にて，ベクトルアーキテクチャと自動ベクトル化コンパイラ，数値シミュレーション言語 DEQSOL 等の研究に従事．平成 4 年から 5 年にかけて米 Purdue 大学計算科学科，独 GMD 客員研究員．平成 9 年 4 月より静岡大学情報学部教授．工学博士．応用数理学会，計算工学会，音響学会，ACM，IEEE 各会員．