ハイパフォーマンス
コンピューティング 62－14
（1996. 8. 28）

レジスタ生存グラフを用いたレジスタ割付け技法

百瀬 浩之† 小松 秀昭‡ 古関 聰† 深澤 良彰†
†早稲田大学理工学部 ‡日本IBM(株) 東京基礎研究所

プロセッサの特性を生かしたコード生成を行なうためにコンパイラによる最適化技術が重要である。その主要な技術の一つにレジスタ割付けがある。レジスタ割付けを行なう際には、プログラムの並列性を落さないように考慮すること及びプログラムの性質を考慮することが重要である。本手法では、プログラムの性質を考慮することにおいて、実行条件を考慮したレジスタ割付け手法を提案する。本手法では、排他的な実行条件を持つ変数を同じレジスタに割り付けることにより、従来のデータフローを考慮したレジスタ割付け手法よりさらに性能の良いレジスタ割付けを実現している。

Register allocation technique using register existence grapth

Hiroyuki Momose† Hideaki Komatsu‡ Akira Koseki† Yoshiaki Fukazawa†

†School of Science & Engineering, Waseda University
‡Tokyo Research Laboratory, IBM Japan, Ltd.

To generate code sequence using processor's characteristics, optimizations by compilers is important. One of the main optimization techniques is the register allocation. When the register allocation is executed, it is important to consider that register allocator doesn't reduce the parallelism in a program and to consider the characteristics of programs. In program's characteristics, we describes a register allocation technique with considering the condition of execution. Our register allocation technique is more effective than existing techniques considering the data flow because the variables that have the exclusive condition of execution are allocated the same register.

1 はじめに

プロセッサの特性を生かしたコード生成を行なうためにはコンパイラによる最適化が重要である。その主要な技術の一つにレジスタ割付けがある。

プロセッサにおいてレジスタの数は限られているので、コンパイラの中間コードで用いられる仮想的なレジスタ（シンボリックレジスタ）を実レジスタにマッピングする必要がある。このマッピングの操作はレジスタ割付け[1, 2, 3, 5]と呼ばれる。レジスタ割付けにおいて考慮する必要があるのは、生存区間が重なるシンボリックレジスタ、つまり、同時刻において生存しているシンボリックレジスタの数であり、これをレジスタの干渉度と呼ぶ。生存区間が重なるシンボリックレジスタは、同じ実レジスタにマッピングできない。従って、レジスタの干渉度が実レジスタ数を越えている場合は、干渉度をプログラム全体を通して実レジスタ数を越えないように低減させなければならない。レジスタ割付けにおいては、次の2点を考慮する必要がある。

1. 命令レベル並列プロセッサを対象としたプログラムの並列性を落さないレジスタ割付け
2. プログラムの性質(コントロールフローとデータフロー)を考慮したレジスタ割付け

1についての背景及び対応は、文献[6]に述べてある。本論文では本手法が2についてどのように対応しているかを中心に述べる。

プログラムの性質を考慮したレジスタ割付けの流れを、以下に示す。古典的な手法として、大域的なレジスタ割付け、プログラム全体を対象とし干渉グラフを作り、このグラフをカラーリングすることににによってレジスタ割付けを行なう手法がある[3]。さらにこれを拡張したものとして、プログラムをタイルと呼ぶループや条件構造などの階層的な部分に分割し、階層的に干渉グラフのカラーリングによるレジスタ割付けを行なう手法がある[2]。さらに、生存区間の重なりを考慮するだけでなくPDG[4]を用いてプログラムのデータ

フローを考慮したタイプのレジスタ割付け手法が提案されている [5]。そこでは、データフローの考慮によって、プログラムの実行時間により影響を与えるシンボリックレジスタに優先的に実レジスタを割り付けることが可能になっている。我々は、さらに実行条件を考慮したレジスタ割付け手法を提案する。これらに加え実行条件を考慮することによって、同時に必要な実レジスタ数をさらに減らすことができる。例えば、ある時刻 t にシンボリックレジスタ A がある条件が T の元で生存し、シンボリックレジスタ B が同じ条件が F の元で生存するならば、A と B は時刻 t において同じ実レジスタに割り付けることができる。このように排他的な実行条件を考慮することは限られたレジスタ資源を効果的に使用するためには非常に有効である。我々は、このようにデータフローとコントロールフローの両方を考慮するためにシンボリックレジスタ間のデータの流れを把握し、制御依存の表現を備えたレジスタ**生存グラフ**と呼ぶグラフを使う。次の章で、レジスタ生存グラフについて説明する。

2 レジスタ生存グラフ

本手法では、シンボリックレジスタ間の干渉関係を把握し、レジスタの干渉度を算出するために、レジスタ生存グラフを用いる (図 1)。

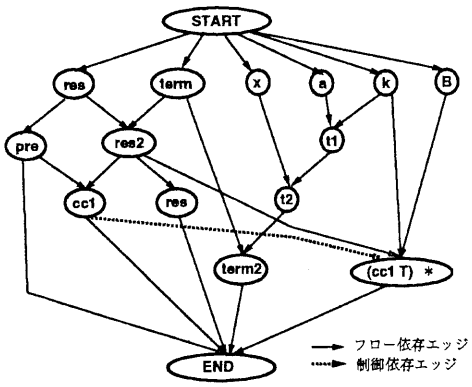

図 1: レジスタ生存グラフ

レジスタ生存グラフは、最内ループに含まれる各シンボリックレジスタをノードとし、シンボリックレジスタの値の使用に関してエッジを張る。例えば、シンボリックレジスタ a の値を使用して新たなシンボリックレジスタ b の値が定義されるならば、a から b へ有向エッジを張る。これをフロー依存エッジということにする。また、ストア命令で使用されるために生存するシンボリックレジスタがあるので、ストア命令のタイミングを表すための**ストアタイミングノード**を導入する (図 1の*)。図 1では、cc1,res2, k,B の全てのシンボリックレジスタが生成された後で、ストア命令が実行されることを意味している。ストア命令で参照されるシンボリックレジスタを表すノードから、ストアタイミングノードに有向エッジを張る。

次に、制御依存を表すためにガードを導入する。実行条件をガード (図 1の cc1 のように、cc+ 条件番号の形式で表す) に代入する。ある実行条件の元で生存するシンボリックレジスタを表すノードには、ガードとその条件の真偽を表す T,F を複数組み合わせたフラグ (cc1 T & cc2 F のように表す) を付加する。生成されたガードを表すノードはガードノードとよぶ。あるガードノードから、そのガードを含むフラグを付加されたノードに向かって有向エッジを張る。この有向エッジが制御依存に相当する。

レジスタ生存グラフの表す最内ループへの入口を表すノードをスタートノード、出口をエンドノードと呼ぶ。最内ループの入口において既に生存しているシンボリックレジスタを表すノードはスタートノードと接続し、ループの出口において生存しているシンボリックレジスタをエンドノードと接続する。スタートノードとノード A の最長パスのノード数をノード A の深さと呼ぶ。各ノードに対し自由度 [5] を定義する。自由度は、最長パスを基準として、各ノードがどれだけ移動できるかを表したものである。また、ノード A がノード B の祖先ノードである時、A は B を支配するという。支配するノードが複数あるノードを分岐点といい、複数のノードから支配されるノードを合流点と呼ぶことにする。分岐点は一つのシンボリックレジスタを参照して複数のシンボリックレジスタが生成されるので、干渉度を増加させるノードである。一方、合流点は複数のシンボリックレジスタを参照して、一つのシンボリックレジスタが生成されるので干渉度を減少させるノードである。

このグラフの利点は、どのシンボリックレジスタ間に干渉がおこりうる可能性があるかを把握しやすいことである。分岐点 A(含むスタートノード) から分岐点 B(含むエンドノード) に至るデータ依存に基づく経路が複数存在する場合、その中の一つの経路上のシンボリックレジスタは、それ以外の経路上のどれか一つのシンボリックレジスタと干渉する。このことを図 1に示したレジスタ生存グラフ上で考えてみる。図 1において、例えば START から t2 に至る経路は START-x-t2 と START-a-t1-t2 の二つの経路があるので、シンボリックレジスタ x は a か t1 と干渉する。

3 レジスタの干渉度の低減

3.1 レジスタの干渉度

ここでは、レジスタ生存グラフを用いてレジスタの干渉度を定義する。レジスタの干渉度とは、ある時間において生存しているシンボリックレジスタの個数である。レジスタ生存グラフにおいては、ノードのみを通過しながらレジスタ生存グラフに干渉する経路を全て横切ったとき、通過したノード数がレジスタの干渉度になる。あるプログラムを表すレジスタ生存グラフのレジスタの干渉度がグラフ全域に渡って実レジスタ数を越えない場合、そのプログラムのシンボリックレ

ジスタはスピルコードを必要とすることなく、実レジスタに割り付け可能である。

この干渉度をグラフ全域について調べるため、レジスタ生存グラフに以下の条件を満たすように横線を入れる。

等レベル線制約
A: 干渉し合う経路は、必ず1回横切る
B: 横線同士が交差しない
C: 全てのノードがどれかの横線で通過されている

同一の線が横切ったノードは、同じ時間に生存することになる。ここでいう時間をレベルと呼ぶことにし、同じレベルを表す線を等レベル線と呼ぶ。各等レベル線に対し、上から順番にレベル番号をふる。また、あるノードxは属性として、xを横切る等レベル線のレベル番号を持つことにする。ある等レベル線の横切ったノードの数をそのレベルのレジスタの干渉度と呼ぶ。本研究では、ロード命令のサイクルを2とし他の命令のサイクルを1と仮定している。ロード命令によってレジスタに読み出されたシンボリックレジスタを表すノードについては、等レベル線は2本横切る必要がある。ストアタイミングノードは、一般のノードと異なった意味合いのノードなので、干渉度を数えるときにカウントしないことにする。また、対象とするアーキテクチャとしては条件フラグレジスタを備えていないものを仮定したので、ガードもレジスタ割付けの対象となる。よって、ガードノードは干渉度を数えるときにカウントするものとする。図2において、グラフの左側にある数字は、そのレベルのレジスタの干渉度である。この例では、全体の最大干渉度が7であるので、このままで、少なくとも7個のレジスタが必要である。

なお、プロセッサの機能ユニット数の制限がないと仮定した時、レジスタ生存グラフに引いた等レベル線の数が実行サイクル数－1に相当する。したがって、実行サイクル数を延ばさないためには、レジスタ生存グラフに引いた等レベル線の数がなるべく増えないように、レジスタの最大干渉度の低減を行なう必要がある。

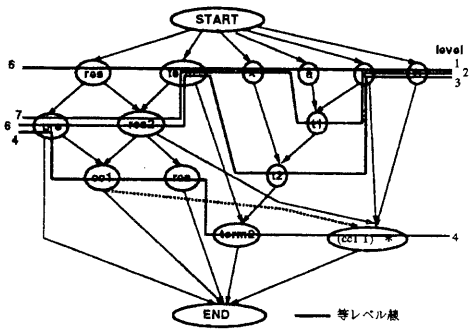

図 2: レジスタの干渉度

3.2 レジスタの最大干渉度の低減

レジスタの最大干渉度の低減操作では、レジスタの最大干渉度が実レジスタ数を越えないようにレジスタ生存グラフを変形する。これは、実行条件を考慮して干渉度を見積もる操作、レジスタ生存グラフにおける分岐点をその分岐点を自分が支配しない合流点の深さまで下げる操作、及びスピルを生成する操作によって行なわれる。

[1] 排他的な実行条件を考慮する

排他的な実行条件の元で生存するシンボリックレジスタは、同じ実レジスタに割り付けることができると考えられる。よって、レジスタ生存グラフにおける各ノードが属性として持つガードの種類とその真偽をみて、排他的な実行条件を持つシンボリックレジスタを考慮してレジスタの干渉度を見積もれば、実行条件を考慮しない場合に比べてレジスタの干渉度を抑えることができる。以下にその手順を示す。

1. 等レベル線の横切るノードを実行条件によって、いくつかの集合に振り分ける。この各集合を実行条件別セットと呼ぶことにする。
2. 実行条件別セットの実行条件が排他的である集合をまとめ、実行条件別セットを要素とした新しい集合を作る。排他的セットと呼ぶことにする。
3. 排他的セット内の実行条件別セットの中でノード数が最大のものを選び、そのノード数を $MaxOfExclusiveSets$ とする。
4. 全ての排他的セットの $MaxOfExclusiveSet$ を合計したものが、実行条件を考慮したレジスタの干渉度となる。

[2] 分岐点を自分が支配しない合流点の深さに下げる

この操作はスピルコードを出さないで、レジスタの干渉度をできるだけ実レジスタ数に近づけるために行なう。一般的に、ある合流点とある分岐点が支配するノードを同一の等レベル線が通過している場合は、分岐点(干渉度を増加させるノード)を合流点(干渉度を減少させるノード)の深さに下げて、分岐点の表すシンボリックレジスタ生存タイミングを遅らせることによって干渉度が低減する。本手法では、この操作をレジスタ生存グラフに仮想依存エッジを追加することによって行なう。仮想依存エッジとは、レジスタの制約を表すために用いられるエッジであり、このエッジのグラフへの挿入は、等レベル線が満たすべき条件に以下を追加する。なお、仮想依存エッジはフロー依存エッジとは違いプログラムの意味を保存するためのエッジなので、等レベル線を自由に横切ることができる。

等レベル線制約
D: 仮想依存エッジの始点ノードを通過する等レベル線は、終点ノード及びそれが支配しているノードを通過してはならない

干渉度を低減するためにある分岐点Aを合流点Bの深さに下げる場合、BからAが直接支配するノードに仮想依存エッジを張る。Bを通過すべき等レベル線がAが支配している複数のノードを通過している場合は、この仮想依存エッジの挿入によりレジスタの干渉度を低減することができる。図3(a),(b)にその例を示す。この例では、分岐点XをXが支配しない合流点Yの深さに下げることによって、3レベル目の干渉度が3から2になっている。なお、この操作に伴ってある実行条件のノードセットが移動した場合には、その実行条件と排他的な実行条件のノードセットは全て同じレベル数だけ移動させなければならない。

[3] スピルを生成する

あるシンボリックレジスタの値をメモリに退避することによって、レジスタの干渉度を低減させることができる。

本手法では、この操作をレジスタ生存グラフにスピルエッジを追加することと、スピルの対象となるノードにマーキングを行なうことによって行なう。スピルエッジとは、仮想依存エッジと同様にレジスタの制約を表すために用いられるエッジであり、このエッジのグラフへの挿入は、等レベル線が満たすべき条件に以下を追加する。

等レベル線制約

E: 論理スピルエッジの始点ノードを通過する等レベル線は、終点ノード及びそれが支配しているノードを通過してはならない

あるシンボリックレジスタAをスピルするためには、まずAをスピルアウトを表すノードとスピルインを表すノードに分割する。スピルアウトノードにはS、スピルインノードにはLのマークをつける。次に、スピルアウトノードから干渉度を低減したいレベルのノードへ、また、干渉度を低減したいレベルのノードからスピルインノードへスピルエッジを張る。図3(c),(d)にその例を示す。この例では、Zをスピルすることによって、2レベル目の干渉度が3から2になっている。なお、スピルエッジも仮想依存エッジと同様にプログラムの意味を保存するためのエッジであるので、等レベル線を引く時には自由に横切ることができる。

図4に、図2に対し、使用可能レジスタ数を5個として、レジスタの最大干渉度を低下させる操作を行なった最終結果を示す。なお、この例においてはBとpreとterm2がスピルの対象になっている。preとterm2には、スピルアウトノードは存在していない。これは、スピルの対象となったシンボリックレジスタがスタートノードにおいて既に生存しているため、本システムが対象としている部分よりも前の部分でスピルアウトを行なうからである。

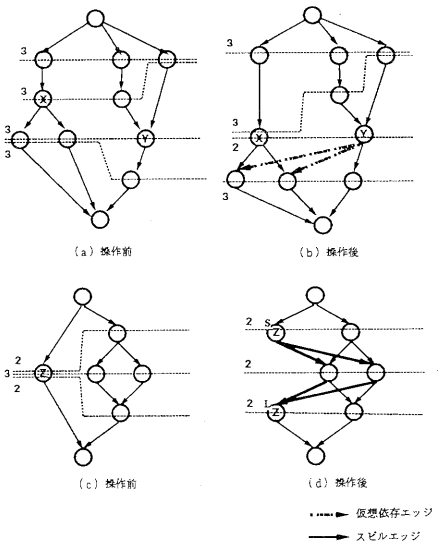

図 3: 干渉度の低減

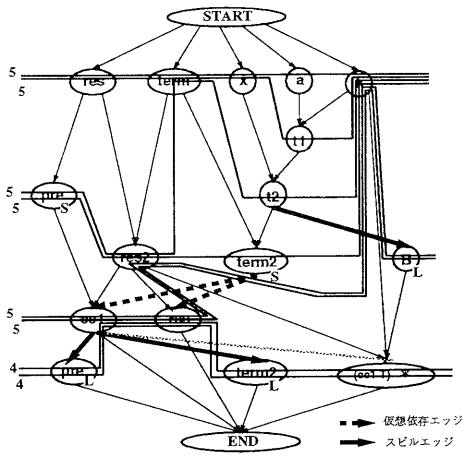

図 4: レジスタの最大干渉度を低下させる操作の結果

4 レジスタ生存グラフを用いたレジスタ割付け

ここまでは、レジスタ干渉度の低減方法について述べてきた。本章では、レジスタ最大干渉度を低減した結果のレジスタ生存グラフを用いて、シンボリックレジスタを実レジスタに実際に割り付ける方法について述べる。この手順においては、以下の項目を考慮することが重要である。

- 前から引き続いて使用されるシンボリックレジスタは同じ実レジスタに割り付けることで、余計なレジスタ転送命令が追加されないようにする。
- レジスタ生存区間の長さを平均化するよりも長いものと短いものに分割した方が、スピルの生成な

どで扱いやすい。

- 新しく実レジスタを使用する際に使用終了時間の最も以前の実レジスタを使用することで、レジスタ再利用の逆依存の影響をできるだけ抑える。

以下に具体的なアルゴリズムを示す。カレントのレベルをiとして、iの初期値を1とする。iのレベルに生存するシンボリックレジスタを全て実レジスタに割り付けるものとする。また、各実レジスタは最も最近使用された際の使用開始レベル及び使用終了レベルの属性を持つことにする。

1. 着目ノードがiより前のレベルから割付けされている場合、割り付けている実レジスタに引き続いて割り付ける。
2. 着目ノードがiのレベルから割付けされる合流点である場合、合流するそれぞれのパスに割り当てられた実レジスタの中で、使用開始レベルの属性が小さい方の実レジスタを選び割り付ける。それ以外の実レジスタはi-1レベルで使用終了となるので、属性に書き込む。
3. 着目ノードが分岐点の子ノードである場合、分岐するそれぞれのパス上の分岐点の子ノードから一つランダムに選び、分岐点が割付けされている実レジスタに割り付ける。残りのパス上の子ノードは、空いている実レジスタの中で使用終了レベルの最も小さいものから順に割付けしていく。
4. iレベルの割付けを全て終了したら、iに1を加え最大の深さをまで上記1から3を繰り返す。

5 評価

スタンフォードベンチマークからいくつかのプログラムを選び、本手法を干渉グラフを使ったレジスタ割付け手法[3]と比較した。表1、表2に、本手法及び従来手法によってレジスタ割付けとコードスケジューリングを行なった場合のクリティカルパスの長さを示す。評価は、固定小数点命令は1サイクル、浮動小数点およびロード／ストア命令は2サイクルで実行が完了するVLIWを用いるものとし、レジスタの数が8,16,32、プロセッサの並列度が2,4,8,∞の場合についてのものである。表より本手法で実行条件を考慮した場合、従来手法に比べかなりの性能向上が得られた。また、実行条件の考慮によって、どれだけ性能向上したかを示すために、図5で本手法の実行条件を考慮した場合、実行条件を考慮しない場合、従来手法の3つを比較する。図5は、レジスタ数を8として、本手法の実行条件を考慮しない場合及び本手法の実行条件を考慮した場合のそれぞれについて、従来手法を1とした時の速度向上の割合を縦軸にとり、棒グラフで表した。排他的な実行条件を考慮した場合を排他的な実行条件を考慮しない場合を比較すると平均では1.05倍の速度向上であったが、ExptabとInitmatでは顕著な速度向上が得られた。これらのプログラムでは実行条件別で生存するシンボリックレジスタの数に偏りがないため、排他的な実行条件の元で生存するシンボリックレジスタを重複して数えていた時と比較して、干渉度をかなり減らすことができたためであると思われる。

6 おわりに

本論文では、プログラムの性質の考慮という面からレジスタ割付けを捉え、従来のデータフローを考慮したレジスタ割付けをさらに進め、プログラムの制御関係を考慮したレジスタ手法を提案した。このために、レジスタ生存グラフというシンボリックレジスタの生存状況を把握し、干渉関係を掴むためのグラフ構造を定義した。また、ガードという表現方法を導入し、レジスタ生存グラフにおける各ノードがどの実行条件の元で生存するかが分かるようになった。この情報を利用し、排他的な実行条件の元で生存するシンボリックレジスタは同じ実レジスタにマッピングすることで、さらに性能の良いレジスタ割付けを行なうことができる。また、文献[6]に述べたように命令レベル並列性も考慮するので、スカラプロセッサから命令レベル並列プロセッサまで幅広く利用できるレジスタ割付け手法となっている。

参考文献

[1] C. Norris and L. L. Pollock. A Scheduler-Sensitive Global Register Allocator. In *International Conference on Supercomputings '93 Proceedings*, November 1993.

[2] D. Callahan and B. Koblenz. Regiser Allocation via Hierarichical Graph Colorling. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, June 1991.

[3] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register Allocation and Spilling Via Graph Colorling. In *Comuter Languages* 1981.

[4] 古関聰、小松秀昭、深澤良彰, 命令レベル並列アーキテクチャのための大域的コードスケジューリング技法とその評価, 並列処理シンポジウムJSPP'94(1994)

[5] 小松秀昭、神力哲夫、古関聰、深澤良彰, 命令レベル並列アーキテクチャのためのレジスタ割付け技法, 情報処理学会論文誌, 2819-30(1995)

[6] 小松秀昭、百瀬浩之、古関聰、深澤良彰, 命令レベル並列プロセッサのためのコードスケジューリング及びレジスタ割付けの協調技法, 並列処理シンポジウムJSPP'96(1996)

表 1: 本手法

レジスタ	8					16					32				
並列度	1	2	4	8	∞	1	2	4	8	∞	1	2	4	8	∞
Bubble	39	22	14	11	11	39	22	14	11	11	39	22	14	11	11
Exptab	67	37	22	18	18	62	35	18	11	10	62	35	18	11	10
Fit	86	63	62	62	62	86	63	62	62	62	86	63	62	62	62
Initarr	79	41	32	32	32	75	39	32	32	32	75	39	32	32	32
Initmat	89	51	29	28	28	89	51	29	28	28	89	51	29	28	28
Permute	33	19	15	15	15	33	19	15	15	15	33	19	15	15	15
Qsort	40	26	19	18	18	38	25	19	17	17	38	25	19	17	17
Remove	81	46	44	44	44	81	46	44	44	44	81	46	44	44	44
Try	109	62	44	42	41	76	61	44	39	39	76	61	44	39	39

表 2: 従来手法 (グラフカラーリングによる方法)

レジスタ	8					16					32				
並列度	1	2	4	8	∞	1	2	4	8	∞	1	2	4	8	∞
Bubble	53	37	37	37	37	39	26	24	23	23	39	26	24	23	23
Exptab	105	91	91	91	91	64	40	35	35	35	62	38	33	33	33
Fit	88	73	70	70	70	86	71	69	69	69	86	71	69	69	69
Initarr	106	96	96	96	96	75	51	49	49	49	75	51	49	49	49
Initmat	150	110	110	110	110	104	67	62	62	62	96	62	58	58	58
Permute	33	25	24	24	24	33	25	24	24	24	33	25	24	24	24
Qsort	46	34	34	34	34	38	25	23	23	23	38	25	23	23	23
Remove	99	83	83	83	83	81	59	57	57	57	81	59	57	57	57
Try	163	137	137	137	137	105	69	66	66	66	105	69	66	66	66

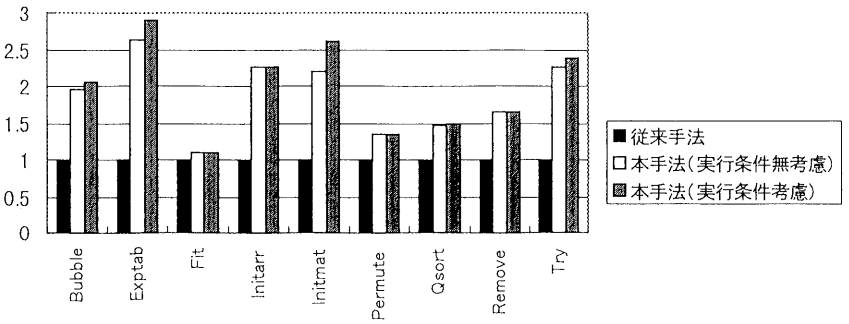

図 5: レジスタ数 8 における比較