非再試行型レジスタ割付けとその評価

片岡正樹[†1] 古関聰[†2]
小松秀昭[†2] 深澤良彰[†1]

グラフ彩色法は，レジスタの多いアーキテクチャにおいては効果的である．なぜなら，スピルコードが出なかった場合，そのレジスタ割付けの結果は最適解だからである．しかし，このアルゴリズムには，レジスタの少ないアーキテクチャにおいて，特に長い処理時間がかかるという問題がある．なぜなら，このアルゴリズムは，干渉グラフが彩色可能になるまで，変数の生存区間分割による干渉グラフの再構築という全過程を繰り返すからである．そこで，我々は，エッジを重み付き双方向エッジにした干渉グラフを用いて，レジスタの少ないアーキテクチャにおける静的コンパイラ向けの，新しい高速なレジスタ割付け手法を提案する．我々の手法は，物理レジスタ数を K とすれば，仮想レジスタを K 色以上の色で塗り分ける．そして，グラフ変形のみでスピルコードを挿入することにより K 色まで色数を減らしている．このようにして，我々の手法は，高速なレジスタ割付けと，生成されたコード内のスピルコード削減を可能にしている．我々の手法を組み込んだコンパイラは，GNU C コンパイラバージョン 3.4.4（gcc-3.4.4）よりも約 8%高速であった．また，スタンフォードベンチマークにおいて，我々のコンパイラが生成したコードは，gcc-3.4.4 の生成したコードより，約 5%高速であった．

Non-retrial Register Allocation and Its Evaluation

Masaki Kataoka,[†1] Akira Koseki,[†2] Hideaki Komatsu[†2]
and Yoshiaki Fukazawa[†1]

Graph coloring is effective for architectures with a large number of registers, because the results of register allocation are optimal if there is no spill code. However, this algorithm has a problem that it particularly takes long time in register allocation for architectures with a small number of registers, because the algorithm iterates the entire process that an interference graph must be rebuilt by spilling live range of a variable until the graph becomes colorable. Thus, we propose a new faster register allocation algorithm in a static compiler for architectures with a small number of registers by using an interference graph with weighted bidirectional edges. Our method applies colors of more than K-colors to virtual registers if K is the numbers of physical registers. Then the number of colors is reduced to K-colors by insertion of spill codes with only transformation of the graph. Thus, our method enables to allocate registers fast and to reduce spill codes in generated codes. A compiler that our method is applied to runs about 8% faster than GNU C Compiler version 3.4.4 (gcc-3.4.4). Also, the codes generated by our compiler run about 5% faster than the codes generated by gcc-3.4.4 in the Stanford Benchmark.

1. はじめに

プログラムを高速実行するためには，コンパイラによる最適化が重要である．その重要な最適化手法の 1 つに，メモリアクセス時間を減少させるレジスタ割付けがある．重要な最適化であるため，今まで多くの研究がされてきたことも事実である．だが，我々は，レジスタの少ないアーキテクチャ向けの，新しいレジスタ割付け手法が必要ではないかと考えている．なぜなら，現在主流なアーキテクチャは，x86 に代表されるレジスタの少ないアーキテクチャであり，レジスタが不足しがちだからである．つまり，対象がレジスタの少ないアーキテクチャの場合，メモリアクセス時間が増加しやすい傾向がある，ということである．

レジスタ割付け手法で有名な手法である，Chaitin らによって提案されたグラフ彩色法[1] は，多くのレジスタ割付け手法の基本アルゴリズムとなっている．この手法では，干渉グラフを作成し，彩色問題を解く

†1 早稲田大学理工学術院
Faculty of Science and Engineering, Waseda University
†2 日本 IBM 株式会社東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

ことでレジスタ割付けを行う．干渉グラフでは，変数はノードで表されている．本論文でいう変数は，中間言語上で値の読み書きが行われるものとレジスタ間のマッピングを行うものという意味で用いる．また，変数が同時に有効である場合には，その変数を表すノードどうしは，無向エッジで接続される．彩色問題を解くときに使用される色数は，対象とするアーキテクチャのレジスタ数と等しい．この色数で干渉グラフを彩色できないときには，変数を 1 つ選択し，その値をメモリに退避するスピルという操作を行う．その後，変数をスピルしたことによって必要となる，干渉グラフを再構築するための全プロセスを，グラフが彩色できるようになるまで繰り返す．ここで，レジスタの少ないアーキテクチャが対象だった場合，彩色に使用できる色数は少なく，この繰返し回数は多くなる．そのため，この手法はレジスタ割付けに時間がかかるので，x86 向け GNU コンパイラコレクション（GCC）のような，静的コンパイラにさえ適していない．

そこで，我々は，レジスタの少ないアーキテクチャでは時間のかかる処理を繰り返さないように，アルゴリズムを変更した．さらに，同時に有効である変数の範囲に着目し，長い時間のかかるメモリアクセスの命令を，できる限り減らそうと考えた．本論文では，高速なコンパイルが可能なだけでなく，高速な実行ファイルも生成可能な新しいレジスタ割付け手法について提案していく．

2. 関 連 研 究

グラフ彩色法は，最も有名なレジスタ割付け手法である．この手法では，レジスタ割付けを彩色問題に置き換えている．レジスタ割付けも，置き換えられた彩色問題も，NP 完全問題であるため，静的コンパイラとして許容される時間で解くために，これまで多くのヒューリスティックスが提案されてきた．さらに，アルゴリズムが単純であるため，レジスタ割付けの基本アルゴリズムとして，多くのコンパイラに採用されてきた．この手法では，まず変数をノードで表し，生存区間の重なっているノードどうしを，無向エッジで接続して，干渉グラフと呼ばれるグラフを生成する．次に，対象とするアーキテクチャのレジスタ数と同じ色数で，この干渉グラフを彩色することを試みる．もし，グラフを彩色できなかった場合には，変数を 1 つ選択し，スピル操作によって，その変数の値をメモリへ退避する．しかし，プログラムを高速に実行するためには，変数が使用されるときに，メモリからレジスタへ，その値を移す必要がある．メモリからレジスタへ値が移された変数は，メモリへ退避される前の変数とは別の変数と考えるため，変数の数が増えることになる．そのため，変数の生存区間などが変更されることになり，変数の生存解析のやり直しと，干渉グラフの再構築を行う必要がある．これらの操作の繰返しは，グラフの彩色が完了するまで行われる．彩色完了後は，色ごとにレジスタを割り当てていくことで，レジスタ割付けが完了する．この手法では，スピル操作が行われなかった場合には，レジスタ割付けの結果が最適であることが保証される．ただし，スピル操作が行われたときには，許容される時間で解くために組み込まれたヒューリスティックスによって，最適解から離れた結果になることが多い．そのため，スピル操作のヒューリスティックスについては，多くの拡張[2)–4)] が提案されてきた．

しかし，レジスタの少ないアーキテクチャでは，彩色に使用できる色数が少ないため，スピル操作の回数は，これらの手法でも多くなってしまう．いい換えれば，グラフ彩色法を基にしたレジスタ割付け手法では，何回も生存区間解析，グラフ再構築，そして彩色を繰り返さなければならない．また，何回スピル操作を行えば，グラフを彩色できるようになるのか分からないという問題もある．

一方，動的コンパイラ向けの，高速なレジスタ割付け手法としては，リニアスキャン[5)] やリザーブドレジスタ方式[6)] などがあげられる．リニアスキャンでは，対象アーキテクチャのレジスタ数の長さを持つアクティブリストと，変数の生存区間情報を持った生存インターバルを利用し，時間ごとにレジスタ割付けを行う．このリストは，ベーシックブロック内のローカルな変数と，それ以外のグローバルな変数とで，定数個ずつ使用できるようになっている．このアクティブリストに入れる操作が，彩色方法と考えることができる．この際，アクティブリストに入りきらなかった変数について，スピル操作を行っている．リザーブドレジスタ方式では，スピルされた変数を読み込むためにしか使わないレジスタをあらかじめ確保しておき，残ったレジスタの数でグラフ彩色法を行う．その際，彩色は 1 度しか行わず，彩色されなかった変数は，すべてスピルするという方法をとっている．いずれも，スピルの際に繰返しをしない方法になっているため，グラフ彩色法に比べ，かなりの速度向上を実現している．

しかし，レジスタ割付けの速度を優先すれば，多くの場合で，レジスタ割付けの性能が犠牲になる．リニアスキャンでは，ヒューリスティックスや特殊レジスタなどが原因となり，グラフ彩色法より 10%ほど性能

が悪くなっている[5]．また，リザーブドレジスタ方式では，レジスタの少ないアーキテクチャが対象の場合は，彩色に使用できる色数がとても少なく，スピルが頻発してしまうため，非常に性能が悪い．どちらも動的コンパイラ向けの手法であるため，速度が重視されるのは仕方ないが，静的コンパイラで使用するならば，性能も重視しなければならない．

3. 本手法の概要

本手法では，レジスタの少ないアーキテクチャが対象の場合でも，グラフ彩色法ほどレジスタ割付けの速度が遅くなく，リニアスキャンほど性能が悪くならない方法を提案する．ここで，各手法の悪い点を改善するためには，干渉グラフの改善，彩色方法の改善，スピル方法の改善の 3 つが必要であると考えた．

まず，干渉グラフの改善であるが，保持している情報を増やす必要があると考えた．今までの干渉グラフでは，ノードが変数のスピルコストを保持し，ノードを接続するエッジについては，あるかないかの 1 ビットの情報しか保持していなかった．そこで，本手法が使用する干渉グラフでは，ノードは変数の生存区間情報を保持できるように，ノードの接続は無向エッジではなく有向エッジを双方向に張るようにし，それぞれの有向エッジに対してスピルコストを保持できるようにしている．これにより，リニアスキャンに近い，高速なレジスタ割付けを行うことが可能になる．本論文では，この操作をエッジコスト付き干渉グラフの生成と呼ぶ．

次に，彩色方法の改善であるが，彩色に使用できる色数を制限する必要があるかどうか考えた．今までの彩色方法では，彩色に使用できる色数は，対象アーキテクチャのレジスタ数以下であった．また，彩色できない場合には，必ずヒューリスティックスによって，彩色できるようにしていた．このヒューリスティックスが，多くの場合で性能低下の原因となっている．そこで，本手法が使用する彩色方法では，彩色できない状態になったら，色数を 1 つ増やすようにしている．このようにすることで，必ず 1 回で彩色が終わることが保証される．本論文では，この操作を過彩色と呼ぶ．

最後に，スピル方法の改善であるが，生存区間解析や干渉グラフ再構築を繰り返す必要があるかどうか考えた．特にグラフ彩色法では，ここでの繰返しが，速度が遅い主な原因になっている．そこで，本手法が使用する彩色方法では，過彩色で増やしてしまった色数を，ノードどうしを合併させることで減らしている．このようにすることで，過彩色で増やした色数と同じ回数だけ，スピル操作を行えばよいことが分かる．本論文では，この操作をスピル付き併合と呼ぶ．

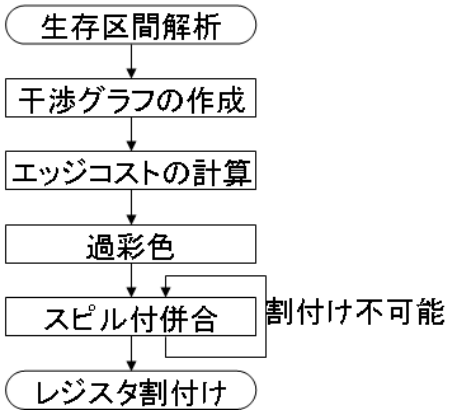

図 1 本手法の流れ
Fig. 1 Flow of our method.

以上の 3 つの考慮を加えた本手法の流れを，**図 1** に示す．それぞれについて，以下詳しく述べる．

3.1 エッジコスト付き干渉グラフの生成

ここでは，ノードが生存区間情報を保持し，生存区間が重なっている変数を表すノード間には，有向エッジが双方向に張られ，それぞれがスピルコストを保持したグラフを作成する．我々は，この有向エッジが保持するスピルコストのことを，エッジコストと呼んでいる．ノードが持つ生存区間情報も，エッジコストも，過彩色やスピル付き併合時に用いることができる．スピルコストが表すものは，2 つの変数を同じレジスタに配置する場合に，必要と見積もられる CPU サイクル数である．エッジコストが双方向である理由は，どちらの変数をスピルするかによって，コストが変わってくるためである．また，ループ内で使用される変数などは，ループを何回繰り返すか分からないことが多く，正確な CPU サイクルの見積りは困難であるが，スピルされにくいように，コストが高めに設定される．ここで，ABI（Application Binary Interface）などによって，特殊レジスタに割り付ける必要がある変数については，特別な考慮が必要である．このような変数を表すノードは，プレファレンス情報などを用いて，あらかじめ彩色される色が限定されるからである．このようなノード間に，エッジが張られていない場合は問題にならないが，エッジが張られていた場合には，エッジが張られないように必要に応じて変数の生存区間分割を行う．

また，生成時の計算量については，従来の干渉グラフと同じで，変数の数を n とすれば，計算量は $O(n^2)$ である．従来の干渉グラフを生成する場合は，縦の長

```
1:        A=E+E
2:        A=A*B
3:        C=A/E
4:        F=1
5:        D=C+A
6:  LOOP:D=D+F
7:        F=F+1
8:        IF F<E GOTO LOOP
9:        D=E/D
```

図 2 サンプルコード
Fig. 2 Sample code.

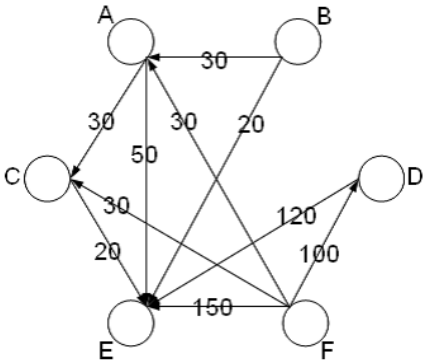

図 3 エッジコスト付き干渉グラフの例
Fig. 3 Example of interference graph with edge-cost.

さが n，横の長さが n のビットフィールドを用意し，生存区間情報から，生存区間の重なっている変数間を表すフィールドに，フラグを立てる方法などがあげられる．本手法の場合は，ビットフィールドの代わりに，2 次元整数配列を用い，フラグを立てる代わりに整数値の加算を用いる．よって，本手法の方が，保持する情報量が多い分，生成時の計算量が大きくなるように感じられるが，実は計算量は同じである．

図 2 のサンプルコードから生成した，エッジコスト付き干渉グラフの例を，**図 3** に示す．本来は，双方向に有向エッジを張り，それぞれにエッジコストを付けるのであるが，ここでは見難くなるため，小さいエッジコストを持った有向エッジのみを示す．

3.2 過 彩 色

ここでは，レジスタ数以上で，かつ，できるだけ少ない色数を用いて，干渉グラフの彩色を行う．基本的には，干渉グラフを何らかの方法で彩色していくことになるが，彩色できなくなった場合には，色数を増やして彩色を続ける方法を用いる．ただし，グラフを彩色できる状態で，色数を増やしてはならない．つまり，用いた彩色方法において，できるだけ少ない色数で彩色を行わなければならない．できるだけ少ない色数でと述べたのは，グラフを彩色できる最小の色数を求める問題は，NP 完全問題だからであり，許容される時間でこの問題を解くためには，何らかのヒューリスティックスが必要になるからである．ただし，対象アーキテクチャにおいて，ここで使用した色数と同じ数のレジスタがあるならば，グラフ彩色法と同様に，彩色結果は最適解であることが保証される．

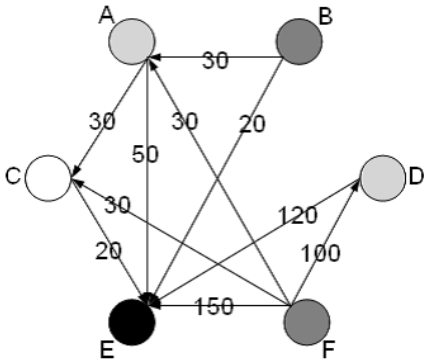

図 4 過彩色の例
Fig. 4 Example of overcoloring.

図 3 の干渉グラフに，レジスタ数を 3 として過彩色を行った例を，**図 4** に示す．

3.3 スピル付き併合

ここでは，過彩色時に増やされた色によって彩色された変数を，スピルを行うことによって他の変数と併合できるようにし，色数をレジスタ数まで減らす作業を行う．いい換えれば，干渉グラフの変形処理のみで，スピル操作を行っている．

スピル付き併合を行う前に，まず干渉グラフ上で，同色で塗られたノードどうしを併合する．干渉グラフのノードが表すものを，変数から仮想レジスタに変換するためである．我々は，この操作を同色ノードの併合と呼んでいる．同色のノードどうしの併合は，エッジが張られていないので，以下のように比較的簡単に行うことができる．ここで，同色の 2 つのノードを選び，それぞれ併合元ノード，併合先ノードと呼ぶことにする．まず，併合元ノードからエッジが張られているノード集合と，併合先ノードからエッジが張られているノード集合の和集合が，併合後のノードのエッジが張られるノード集合になる．そして，あるノード x と併合元ノードとのエッジコストと，ノード x と併合先ノードとのエッジコストの和が，ノード x と併合後のノードとのエッジコストとなる．このとき，同じ方向のエッジどうしで，エッジコストの和をとるようにする．同色ノード併合が終了すると，干渉グラフは彩色で用いた色数と同数のノードがある，完全グラフに変形される．

図 4 の干渉グラフに，同色ノード併合を行った例を，**図 5** に示す．

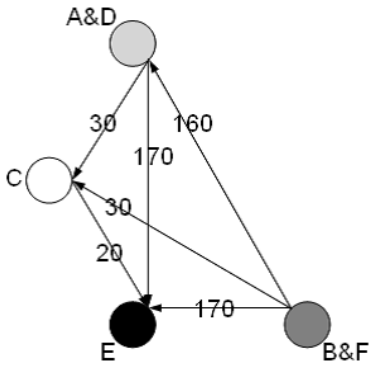

図 5 同色ノードの併合例
Fig. 5 Example of coalescing nodes of same color.

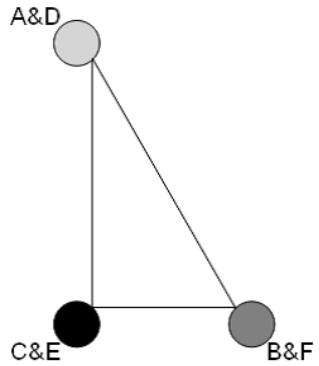

図 6 スピル付き併合の例
Fig. 6 Example of coalescing with spilling.

ここからさらに，干渉グラフを彩色するために増やした色数と同数のノードを併合によって減らす．まず，スピルにかかるコストを示すエッジコストが，最も小さいエッジを探索し，選択する．図 5 の例では，エッジコストが 20 と一番小さい，ノード C からノード E へのエッジが選択される．ここで，エッジの始点にあるノードを併合元ノード，エッジの終点にあるノードを併合先ノードと呼ぶことにする．つまり，例の場合では，ノード C が併合元ノード，ノード E が併合先ノードである．この併合元ノードと併合先ノードを，スピルコードを出しながら併合することが，スピル付き併合である．スピルコードの出し方は，併合先ノードが表す変数をまずスピルし，併合元ノードの変数がレジスタに載るようにする．そして，併合先ノードが表す変数が必要になる場合には，併合元ノードが表す変数と入れ替えを行うようにする．また，このスピルを行う区間は，併合先ノードが表す変数と，併合元ノードが表す変数の生存が重なっている区間だけでよい．この変数の入れ替えのタイミングや，生存の重なっている区間を求めるために，併合先ノードと併合元ノードが保持する生存区間情報が用いられる．そして，併合先ノードと併合元ノードの保持する生存区間情報を重ね合わせ，生存の重なった区間にスピルの情報を加えた生存区間情報が，併合後ノードの保持する生存区間情報となる．例の場合では，変数 E がスピルされ，変数 C がレジスタに載るようなコードが出される．次に，変数 C の生存が終了する図 2 の 5 の命令の後に，変数 E のスピルインコードが挿入される．そして，併合後の変数 C&E の生存区間情報が，変数 C と変数 E から生成される．変数 C&E の生存区間情報は，図 2 の 1 の命令以前から変数 E が生存しており，3 の命令から変数 C が生存し，5 の命令の次に挿入されたスピルインコードからまた変数 E が生存するという情報になっている．また，エッジコストの処理については，併合元ノードからエッジが張られているノード集合と，併合先ノードからエッジが張られているノード集合の和集合から，併合元ノードと併合先ノードを除いた集合が，併合後のノードのエッジが張られるノード集合になる．そして，併合後のノードとエッジでつながっているノードとのエッジコストを再計算する．このスピル付き併合を繰り返し，対象アーキテクチャのレジスタの個数と同数のノードを持つ完全グラフになったときが終了状態である．例の場合では，この終了状態にあたるため，エッジコストの再計算を行う必要はない．

図 5 の干渉グラフに，レジスタ数を 3 としてスピル付き併合を行った例を，**図 6** に示す．

4. 実　　験

ここでは，性能の比較実験を行う．実験環境は，CPU がインテル Core2Duo プロセッサ 2.66 GHz，メモリ 2 GB，OS が Windows XP SP2 のマシンを用いた．本手法を実装するために用いたコンパイラは，Cygwin gcc-3.4.4 という C コンパイラで，レジスタ割付けの部分のみ本手法に書き換えて実験を行った．比較対象としては，オリジナルの Cygwin gcc-3.4.4 を用い，コンパイルにかかった時間と，生成されたコードの実行時間を比較した．オリジナルの Cygwin gcc-3.4.4 を選んだ理由は，x86 用のレジスタ割付けにリニアスキャン[5] を用いているからである．実験に用いたベンチマークは，Stanford Benchmark である．**図 7** に，コンパイル 1 回にかかった平均時間を，プログラムごとにミリ秒で，**図 8** に，生成されたコードの 1,000 回分の実行時間の比率を，オリジナルを 1 としてプログラムごとに示す．つまり，図 7 でも図 8 でも，棒の短い方がより高速であることを示している．

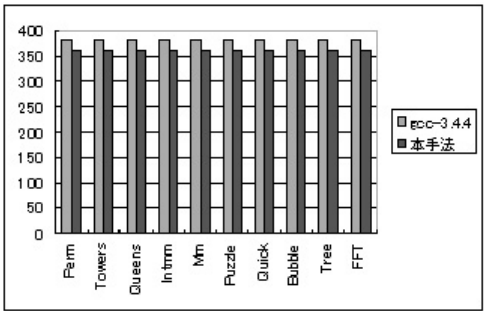

図 7 コンパイルにかかった平均時間（ミリ秒）
Fig. 7 Average time (millisecond) required for compiling.

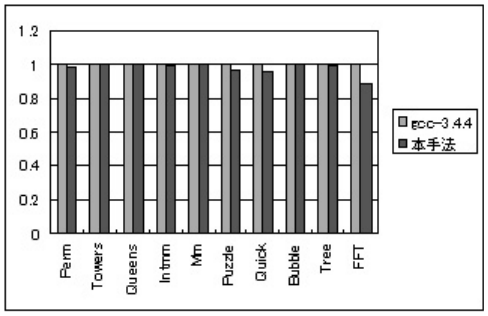

図 8 生成コード 1,000 回実行の時間比率
Fig. 8 The ratio of execution time for 1,000 times of generated codes.

このように，Cygwin gcc-3.4.4 と比較して，レジスタ割付け時間が短くなった分，コンパイル時間の短縮に成功し，生成されたコードの性能についても良くなっているか，ほぼ同等という結果となった．本手法は，Puzzle のようなループ構造を多く持ったプログラムや，Tree のような末尾再帰があるプログラムに，効果的であるということがいえると思われる．

今回，GCC の RTL から生存区間解析を行っているため，変数の大域的な生存区間分割や，SSA 変換[8]などは，GCC 任せになっている．また，value numbering[7] といった手法を組み込めば，より高い性能が出る可能性がある．

5. 終わりに

本論文では，静的コンパイラ向けの，高速でかつ性能もあまり低下しないレジスタ割付け手法を提案した．実験結果では，gcc-3.4.4 より，8%ほどの速度向上と，平均 5%ほどの性能向上が見られた．この実験結果は，gcc-3.4.4 が用いているレジスタ割付け以外の最適化手法を用いた場合の結果である．そのため，gcc-3.4.4 が用いていない最適化との相性は，今回の実験では検証されていない．どのような最適化手法と協調性があるか，また，どれほどの性能が得られるかどうかは，今後の実験で検証したいと考えている．たとえば，生存区間分割において，変数の使用頻度の高い区間と低い区間を，計算量を上げずに切り離すことが可能になれば，今以上の実行性能の向上が見込めると思われる．また，より良い彩色方法の検討もしたいと考えている．

参 考 文 献

1) Chaitin, G.J., Auslander, M.A., Chandro, A.K., Cocke, J., Hopkins, M.E. and Markstein, P.W.: Register Allocation via Coloring, *Computer Languages*, Vol.#6, pp.47–57 (1981).
2) Norris, C. and Pollock, L.L.: A Scheduler-Sensitive Global Register Allocation, *Proc. ACM SIGPLAN '93 Conf. on Supercomputing*, pp.804–813 (1993).
3) Pinter, S.S.: Register Allocation with Instruction Scheduling: A New Approach, *Proc. ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pp.248–257 (1993).
4) Bergner, P., Dahl, P., Engebretsen, D. and O'Keefe, M.: Spill Code Minimization via Interference Region Spilling, *Proc. ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation*, pp.287–295 (1997).
5) Poletto, M. and Sarkar, V.: Linear scan register allocation, *ACM TOPLAS*, Vol.21, No.5, pp.895–913 (1999).
6) Koseki, A., Komatsu, H. and Nakatani, T.: Spill Code Minimization by Spill Code Motion, *12th International Conf. on Parallel Architectures and Compilation Techniques* (*PACT'03*), pp.125–134 (2003).
7) Briggs, P.: Value Numbering, *Software: Practice and Experience*, Vol.27, No.6, pp.701–724 (1997).
8) Cytron, R., Ferrante, J., Rosen, B., Wegman, M. and Zadeck, K.: An Efficient Method of Computing Static Single Assignment Form, *Conf. Record of the 16th ACM Symposium on the Principles of Programming Languages*, pp.25–35 (1989).

(平成 19 年 7 月 9 日受付)
(平成 19 年 10 月 12 日採録)

片岡　正樹

1980 年生．2004 年早稲田大学大学院理工学研究科情報科学専攻修了．同年同研究科コンピュータ・ネットワーク工学科博士課程進学．2005 年早稲田大学理工学部助手就任，現在に至る．コンパイラの最適化技術に関する研究に従事．

古関　　聰（正会員）

1969 年生．1998 年早稲田大学大学院理工学研究科電気工学専攻博士課程修了．同年日本 IBM（株）入社．以来，同社東京基礎研究所において，Java Just-in-Time コンパイラの開発に従事．工学博士．ACM 会員．

小松　秀昭（正会員）

1960 年生．1985 年早稲田大学大学院理工学研究科電気工学専攻修了．同年日本 IBM（株）東京基礎研究所入社．コンパイラ，アーキテクチャ，並列処理の研究に従事．博士（情報科学）．

深澤　良彰（正会員）

1953 年生．1976 年早稲田大学理工学部電気工学科卒業．1983 年同大学大学院博士課程修了．同年相模工業大学工学部情報工学科専任講師．1987 年早稲田大学理工学部助教授．1992 年同教授．2007 年同大学基幹理工学部教授．工学博士．ソフトウェア工学，コンピュータアーキテクチャ等の研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE，ACM 各会員．