2007－ARC－174 (16)
2007／8／1

生存区間分割時に発生する偽干渉を避けるための同時コピー中間コードの利用

中林 淳一郎[†] 片岡 正樹[††] 古関 聰[†††]
小松 秀昭[†††] 深澤 良彰[††]

グラフ彩色法はレジスタ割付けの代表的な手法であるが，冗長なスピルが発生しやすいという欠点があった．この冗長なスピルの発生を抑えるために，生存区間分割という手法が提案されている．しかし，従来の手法では生存区間分割を行った際に余計な干渉が発生し，最適なレジスタ割付けを妨害してしまう場合があった．本論文では，同時コピー中間コードを定義して利用することで余計な干渉の発生を回避し，最適なレジスタ割付けを可能にする生存区間分割手法を提案する．実験の結果，本手法を実装したコンパイラは，本手法未実装のコンパイラに対して最大6%程度実行速度が速いコードを出力可能なことが確認できた．

Using Concurrent-copy Intermediate Code To Avoid Pseudo-interference Generated In Live-range Splitting

Junichiro Nakabayashi ,[†] Masaki Kataoka ,[††] Akira Koseki ,[†††]
Hideaki Komatsu [†††] and Yoshiaki Fukazawa [††]

Graph coloring is a typical technique of register allocation. But it has a problem to generate redundant spill codes easily. To solve this problem, some techniques of live-range splitting have been proposed. However, previous techniques generate pseudo-interference that disturb optimal register allocation. We propose a new technique of live-range splitting that can avoid to generate pseudo-interference by defining and using concurrent-copy intermediate code in this paper. Our experimental results have shown that our technique improves the performance of a compiler by maximum of about 6%.

1. はじめに

コンパイラにおけるレジスタ割付けの代表的な手法として，グラフ彩色法[1]が挙げられる．グラフ彩色法では，干渉グラフと呼ばれる無向グラフを生成し，そのグラフに対して彩色問題を解くことでレジスタ割付けを行う．

しかしながら，単純なグラフ彩色法では最適なスピルが行えない場合がある．例えば，スピル対象として選択された変数はその生存区間全域においてスピルされるため，実際にはレジスタに空きがある区間においてもスピルコードが発行されてしまう．

† 早稲田大学大学院基幹理工学研究科
Graduate School of Fundamental Science and Engineering, Waseda Univ.
†† 早稲田大学理工学術院
Faculty of Science and Engineering, Waseda Univ.
††† 日本 IBM（株）東京基礎研究所
Tokyo Research Laboratory, IBM Japan, Ltd.

このようなグラフ彩色法の問題点を解決するためには，生存区間分割[2)~4)]という手法が必要である．生存区間分割とは，変数の生存区間を複数の小区間に分割し，それらの小区間ごとに別のレジスタを割付けたりスピルしたりすることを可能にする手法である．これにより，前述のような冗長なスピルの発生を抑えることができる．

ただし，生存区間分割には悪影響も存在する．その1つが，生存区間分割を行うことで連続的なコピーが挿入され，本来存在しなかったはずの余計な干渉が発生してしまうことである．以下では，この余計な干渉の事を偽干渉と呼ぶことにする．偽干渉が発生することで変数の干渉度が高まり，最適なレジスタ割付けを行えない場合がある．例えば，生存区間分割を行っていない状態であれば干渉度がレジスタ数以下であり，スピルを行う必要がなかった区間であっても，分割を行うことで干渉度が高まり，スピルが必要になってしまうということがある．

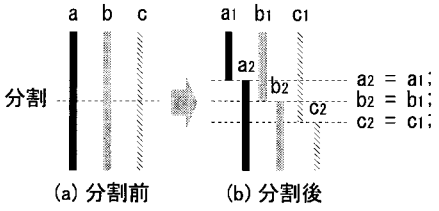

図 1 偽干渉が発生する例

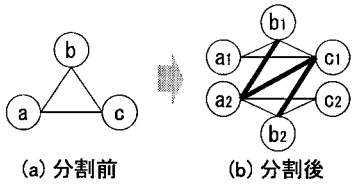

図 2 偽干渉の例

生存区間分割を行った際に偽干渉が発生する例を**図 1** に示す．また，そのときの干渉グラフの様子を**図 2** に示す．図 2 において，太線で描かれている部分が偽干渉である．制御フローグラフのエッジにおいて生存区間分割を行った場合，同時に複数の変数の生存区間を分割することになるため，分割する変数の数だけコピー命令が必要になる．但し，複数のコピー命令を同時に実行することはできないので，1 つずつ処理していくことになり，連続的なコピーが発生する．そして，連続的なコピーは偽干渉を発生させる．例えば図 2 の (b) で，a_2 は b_2 と c_2 の他に，b_1 や c_1 とも干渉している．これは，コピー操作が連続的に行われるために，a_2 の生存区間が開始する点においては b_1 や c_1 も生存している必要があるからである．このような干渉は本来存在しなかったものであり，偽干渉であるといえる．

このように，複数のコピー命令を同時に実行できないために，偽干渉が発生する．つまり，複数のコピー命令を同時に実行できると仮定した上で生存区間分割およびレジスタ割付けを行うことができれば，偽干渉は発生しない．

そこで，本論文では，複数のコピー命令を同時に実行することを表す同時コピー中間コードを用意し，それを利用した生存区間分割手法を提案する．これにより，偽干渉の発生を防ぐことできるので，最適なレジスタ割付けが可能になる．また，本手法を実装したコンパイラを用意し，従来のコンパイラと比較することで本手法の有効性を検証する．

2. 関連研究

代表的なレジスタ割付け手法として，グラフ彩色法[1] がある．しかしながら，単純なグラフ彩色法ではスピル操作を行う際に各生存区間の局所的な性質を考慮しないため，最適なスピルが行えない場合がある．例えば，一部の区間においては干渉度が高くてレジスタが不足するが，その他の区間においては干渉度が低くてレジスタに空きがあるという生存区間を持つ変数 v について考える．v がスピル対象として選択された場合，単純なグラフ彩色法では v の生存区間全域をスピルするため，レジスタに空きがある区間に関してもスピルコードを発行してしまう．しかし，実際は干渉度が高い区間のみスピルすれば，その他の区間に関してはレジスタ上に置いておくことが可能であり，これが最適なスピル操作である．

上記のようなグラフ彩色法の問題点を解決するために，生存区間分割という手法がある．生存区間分割とは，変数の生存区間を複数の小区間に分割し，それぞれ別々の変数として扱うことで生存区間の局所的な性質を考慮できるようにする手法である．これにより，生存区間の一部のみをスピルすることが可能になり，冗長なスピルの発生を抑えることができる．

生存区間分割については，分割の仕方等が異なるいくつかの手法が提案されている．Briggs は，静的単一代入形式[5] (SSA 形式) への変換や SSA 逆変換において ϕ ノードへ至るエッジや逆 ϕ ノードから出るエッジで分割を行うという手法を提案した[2]．Kolte と Harrold は，load/store ranges と呼ばれる，Briggs の手法よりも小さな区間に分割する手法を提案した[3]．load range とは，少なくとも 1 つの use を含む小区間であり，store range とは，少なくとも 1 つの def を含む小区間である．また，Nakaike らは，制御フローが合流したり分岐したりする点で分割を行い，その後プロファイル情報を用いて，実行頻度が高い区間に挿入されてしまったコピー命令を削除するという手法を提案した[4]．特に Nakaike らが行ったような，制御フローのエッジにおいて生存区間分割を行う手法は効果が高い．干渉度の高低や実行頻度の高低といった局所性を考慮できるため，冗長なスピルの発生を回避しやすいからである．

これらの手法では，分割によって発生した余計なコピー命令を削除するために，コアレシング[6][7] と呼ばれる処理を必要としている．生存区間分割には，大量のコピー命令を挿入することでプログラム実行速度の低下を招いてしまうという悪影響があるため，余計な

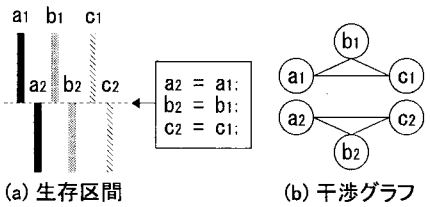

図 3 同時コピーを用いた生存区間分割の例

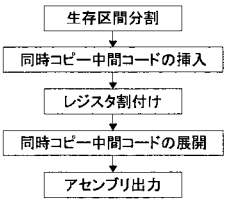

図 4 本手法の流れ

コピー命令は除去しなければならないからである．

しかし，生存区間分割には，コアレシングでは解決することのできないもう 1 つの悪影響が存在する．それは，分割を行うことで偽干渉が発生し，干渉度が高まることで最適なレジスタ割付けを行えない場合があるというものである．制御フローのエッジにおいて生存区間分割を行うと連続的なコピーが挿入され，それによって偽干渉が発生する．そのため，コアレシングでは偽干渉の発生を回避することができない．生存区間分割の効果を得るためには，必ずいくつかのコピー命令を残しておかなければならないからである．よって，偽干渉の発生を回避しつつ余計なコピー命令を削除することができるような，コアレシングに代わる手法が生存区間分割には必要である．

そこで，本論文では，生存区間分割およびレジスタ割付け時に，複数のコピー命令を同時に実行することができると仮定して処理し，レジスタ割付け完了後に実際のプロセッサが処理できる形に戻すことで，この問題を解決する手法を提案する．

3. 本手法の特徴

前章で述べたように，制御フローのエッジで生存区間分割を行う手法には，偽干渉を発生させてしまうという問題点があった．そこで，我々は偽干渉を発生させない生存区間分割を提案する．これにより，生存区間分割の効果を十分に引き出すことができ，最適なスピル操作が可能になる．

前述のように，偽干渉の発生を防ぐためには複数のコピー命令を同時に実行できればよい．コピー命令を同時に実行できると仮定した場合の生存区間分割の例を**図 3** に示す．図 2 の (b) と比較することで，図 3 の状況では偽干渉が発生していないことがわかる．

しかし，現在のプロセッサでは複数のコピー命令を 1 サイクルで全く同時に行うことは不可能である．そこで，我々は複数のコピー命令を同時に実行することを表す同時コピー中間コードを用意し，それを利用することで，コピー命令を同時に実行できると仮定して処理できるようにした．

本手法の流れを**図 4** に示す．以下では，各ステップの詳細を説明していく．

3.1 生存区間分割

レジスタ割付けを行う前に，まず生存区間分割を行う．生存区間分割の手法については前述のようにいくつかの提案がなされているが，本手法では簡単のために，制御フローにおけるループ構造の前後で分割を行うという手法をとることにした．ループ構造内の処理はループ構造外の処理に比べて実行頻度が高い．よって，その前後で分割を行うことで実行頻度の高い区間と低い区間を区別することができる．これにより，実行頻度の高い区間，低い区間でそれぞれ別々の生存区間をスピル対象として選択できるので，冗長なスピルの発生を抑えることができる．

3.2 同時コピー中間コードの挿入

生存区間分割を行った後に，分割によって発生した連続的なコピー命令をそれぞれ一纏めにし，それらを 1 つの同時コピー中間コードで置き換える．同時コピー中間コードには複数のコピー命令が含まれており，それらのコピー命令は全く同時に実行されることを意味する．

3.3 レジスタ割付け

レジスタ割付けの基本的なアルゴリズムはグラフ彩色法である．但し，生存区間解析および干渉グラフ生成の際には，同時コピー中間コードに対して特別な配慮が必要になる．それは，同時コピー中間コードに含まれているコピー命令は同時に実行されるものとして扱わなければならないということである．そして，それらのコピー命令の左辺変数の生存区間は同時コピー中間コードの地点から開始され，右辺変数の生存区間はその地点で終了するように設定しなければならない．こうすることで，同時に実行されるコピー命令の右辺変数と左辺変数の間には干渉が発生せず，すなわち，偽干渉が発生しないことになる．偽干渉が発生しなければ，最適なスピル操作が可能になる．

3.4 同時コピー中間コードの展開

レジスタ割付けが完了すれば，その次にはアセンブ

リコードの出力が行われる．但し，アセンブリコードを出力する際には同時コピー中間コードの存在は許されない．なぜなら，現在のプロセッサでは複数のコピー命令を全く同時に実行することは不可能であり，同時コピー中間コードに対応するマシンコードは用意されていないからである．

よって，レジスタ割付けが完了した時点で同時コピー中間コードを削除し，そこに含まれていたコピー命令を適切な順番に並べて，同時コピー中間コードがあった地点に挿入する必要がある．この操作を同時コピー中間コードの展開と呼ぶことにする．

同時コピー中間コードを展開する際には，特にコピー命令の並び順に注意して展開しなければならない．レジスタ割付けの間はそれらのコピー命令は同時に実行されると仮定して処理しているため，誤った順番に並べてしまうと同時に生存している変数がレジスタ数を超えてしまうという状況が発生することがあるからだ．

同時コピー中間コードを展開する際のルールを以下に列挙する．

(1) 左辺変数がスピルされているコピー命令は，コピー命令を並べる際に先頭に配置する
(2) 右辺変数がスピルされているコピー命令は，コピー命令を並べる際に最後尾に配置する
(3) 左辺変数と右辺変数が共にスピルされているコピー命令は削除し，それらの変数は分割前の状態に戻す
(4) 左辺変数と右辺変数に同じレジスタが割付けられているコピー命令は削除し，それらの変数は分割前の状態に戻す
(5) 左辺変数と右辺変数に異なるレジスタが割付けられているコピー命令は，他のコピー命令との依存関係を考慮し，適切な順番に並べる

以下では，これらのルールの詳細を説明していく．

3.4.1 スピルされた変数を含むコピー命令

まず，左辺変数あるいは右辺変数がスピルされているコピー命令に注目し，それらを同時コピー中間コードから抜き出して適切な位置に配置する．左辺変数がスピルされているコピー命令は，同時コピー中間コードよりも先に実行されなければならない．同時コピー中間コードよりも後に実行されるように配置すると，右辺変数の生存区間が偽干渉を発生させてしまうためである．よって，左辺変数がスピルされているコピー命令は，命令リストにおいて同時コピー中間コードの前に配置する．逆に，右辺変数がスピルされているコピー命令は，同時コピー中間コードの後に配置する必要がある．

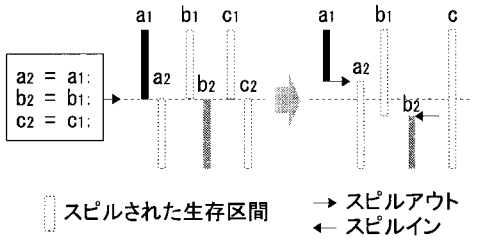

図 5 スピルされた変数を含むコピー命令の展開

左辺変数がスピルされているコピー命令は，スピルアウト操作に変換される．このとき，左辺変数の値を格納するために用意されたメモリ領域に対して，右辺変数の値を直接書き込むようにすることで，コピー命令は必要なくなり，削除することができる．同様に，右辺変数がスピルされているコピー命令は，スピルイン操作に変換される．

ちなみに，左辺変数と右辺変数が共にスピルされているコピー命令は，同時コピー中間コードから削除し，その変数名を統一する．そのようなコピー命令はメモリ上の変数からメモリ上の変数へ無意味なコピーを行うだけなので，削除してしまっても問題がない．

以上のような，スピルされた変数を含むコピー命令を展開する様子を図 5 に示す．図 5 において，a に関するコピー命令は左辺変数がスピルされているコピー命令であり，b に関するコピー命令は右辺変数がスピルされているコピー命令である．そして，c に関するコピー命令は左辺変数と右辺変数が共にスピルされているコピー命令である．

3.4.2 左辺と右辺に同じレジスタが割付けられたコピー命令

左辺変数と右辺変数に同じレジスタが割付けられたコピー命令は，あるレジスタから同じレジスタにその値をコピーするという全くの無駄な処理であるため，同時コピー中間コードから削除する．

3.4.3 左辺と右辺に異なるレジスタが割付けられたコピー命令

ここまでの処理を順番に行っていくと，同時コピー中間コードが含んでいるコピー命令は，左辺変数と右辺変数に異なるレジスタが割付けられたコピー命令のみになっているはずである．これらのコピー命令を抜き出して適切な実行順序に基づいて並べ，同時コピー中間コードと置き換えることで同時コピー中間コードの展開は完了する．

但し，これらのコピー命令の間には依存関係が存

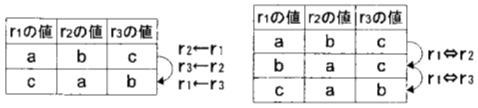

図 6 スワップ操作の利用例

在する場合があり，並び順を決定する際にはその依存関係に基づいた順番にしなければならない．例えば，r_1，r_2，r_3 をそれぞれレジスタとするとき，$r_2 \leftarrow r_1$，$r_3 \leftarrow r_2$ という 2 つのコピー命令が同時コピー中間コードに含まれていたとする．ここで，$r_2 \leftarrow r_1$ を実行してから $r_3 \leftarrow r_2$ を実行する場合と，$r_3 \leftarrow r_2$ を実行してから $r_2 \leftarrow r_1$ を実行する場合では，最終的な r_3 の値が異なる結果になる．この場合，これらのコピー命令は同時に実行されると仮定しているため，プログラムが想定している正しい結果を得るには後者の実行順序でなければならない．このように，実行する順序によって結果が異なってしまうためにコピー命令の並べ方に制約が発生する状況を，これらのコピー命令間には依存関係があるという．

さらに特殊な状況として，コピー命令間の依存関係がループしている場合がある．例えば，(1)$r_2 \leftarrow r_1$，(2)$r_3 \leftarrow r_2$，(3)$r_1 \leftarrow r_3$ という 3 つのコピー命令があるとする．(1) と (2)，(2) と (3)，(3) と (1) の間にはそれぞれ依存関係があり，(1) より先に (2) を，(2) より先に (3) を，(3) より先に (1) を実行しなければならないという制約が発生する．よって，依存関係がループしている状況では実行順序を定めることができない．

このような状況では，スワップ操作を利用して解決することにした．その例を**図 6** に示す．図 6 の (a) は，3 つのコピー命令を同時に実行した場合の各レジスタ上のデータの様子を示している．(b) はスワップ操作を利用して，各レジスタ上のデータが最終的に (a) と同じ状態になるようにデータを移動させた様子を示している．2 回のスワップ操作によって，コピー命令を同時に実行した場合と同じ状態を再現できることがわかる．このように，依存関係がループしているコピー命令はスワップ操作に置き換えることで，プログラムが想定している正しい動作を実現できる．

しかし，まだレジスタ間のスワップ操作をどのように実現するか，という問題が残っている．この問題は，XOR 演算を用いて解決することにした．例えば，r_1，r_2 という 2 つのレジスタ上のデータをスワップすることを考える．この場合，まず $r_1 \leftarrow r_1 XOR r_2$ を実行し，次に $r_2 \leftarrow r_1 XOR r_2$ を実行し，最後にもう一度 $r_1 \leftarrow r_1 XOR r_2$ を実行する．これにより，r_1 と r_2 が保持しているデータを入れ替えることができる．

このように，コピー命令間の依存関係がループしている場合は，それらのコピー命令を XOR 演算によるスワップ操作に置き換えることで同時コピー中間コードを展開することができる．しかし，この方法を実際に使用すると命令数の増加を招いてしまう．スワップ操作を 1 回行うためには 3 回の XOR 演算が必要なため，上記の例のような 3 つのコピー命令は 6 回の XOR 演算に置き換えられることになるからである．但し，XOR 演算の処理はプロセッサ内で完結しているため，本手法を用いないことで発生するメモリアクセスのオーバーヘッドと比較すれば，XOR 演算の増加によるオーバーヘッドは無視できるものである．

以上で，同時コピー中間コードの展開は完了する．

4. 実験と評価

本手法の有効性を示すために以下の 3 つのコンパイラを用意し，Dhrystone ベンチマークを用いて実験を行った．

(1) COINS コンパイラ[8] (Ver. 1.4.2)
(2) 同時コピー中間コードを利用しない生存区間分割手法を実装した COINS コンパイラ
(3) 本手法を実装した COINS コンパイラ

COINS コンパイラのレジスタ割付けにはグラフ彩色法が利用されており，本手法の実装および比較が容易であったために，これを用いた．(2) のコンパイラの生存区間分割アルゴリズムは本手法で用いたものと全く同じである．すなわち，制御フローのループ構造の前後で分割を行うというものである．(2) のコンパイラと (3) のコンパイラで異なるのは，同時コピー中間コードを利用しているか否かという点のみである．さらに，(2) と (3) のコンパイラに関しては，分割を行うか否かを決定する条件を若干変化させたものを用意した．その条件とは，ループ構造内の干渉度が x 以上であれば分割を行い，それ以下であれば分割を行わない，というものである．干渉度が低い区間ではスピルを行う必要はないので，分割を行っても意味はなく，無駄にコピー命令が増加するだけである．そのため，干渉度がある程度高い区間でのみ分割を行った方が性能向上につながりやすいと考えられる．以下では，この条件を”閾値 x”という形式で示す．例えば，閾値 0 とはループ構造内の干渉度が 0 以上の場合，すなわち，すべてのループ構造において分割を行うことを示し，閾値 4 とはループ構造内の干渉度が 4 以上の場合にそのループの前後で分割を行うことを示す．今回は

表 1 Dhrystone をコンパイルした際に発生する干渉数の比較

閾値	COINS	生存区間分割のみ実装	本手法実装
0	414	636	556
4	414	619	546
5	414	549	504
6	414	479	454

表 2 Dhrystone の実行結果（Dhrystone 数/秒）の比較

閾値	COINS	生存区間分割のみ実装	本手法実装
0	192281	191028	202862
4	192281	192131	203876
5	192281	190048	197174
6	192281	194794	201582

閾値 0 および閾値 4〜6 のものを用意して実験を行った．閾値 1〜3 のものを用意しなかった理由は，干渉度が 1〜2 であるループ構造は Dhrystone には存在せず，閾値 0 とした場合と結果が同じになるからである．

まず，上記 3 つのコンパイラでそれぞれ Dhrystone ベンチマークをコンパイルし，レジスタ割付けフェーズで最初に干渉グラフを生成した際の総干渉数，すなわち干渉グラフにおけるエッジの数を比較した．その結果を**表 1** に示す．

この結果から，生存区間分割を実装したコンパイラと本手法を実装したコンパイラは，閾値が高くなるほど干渉数が減っていく様子がわかる．これは，閾値が高くなるにつれて，分割を行う箇所が少なくなっていくためである．また，本手法を実装したものは，同時コピー中間コードを用いない生存区間分割を実装したものに比べて干渉数が少なくなっていることがわかる．これは，本手法によって偽干渉の発生が回避されたためであり，この干渉数の差が偽干渉の数を示していると考えられる．

次に，上記 3 つのコンパイラが出力した実行ファイルをそれぞれ 10 回ずつ実行し，得られたベンチマーク結果の平均値を比較した．実行環境は，CPU が PentiumM（1GHz），メモリ 512MB，OS は WindowsXP Professional である．その結果を**表 2** に示す．Dhrystone ベンチマークは計測結果として 1 秒間の Dhrystone 数（1 秒間にメインループを何回回ったか）を出力する．表 2 の数値はこの Dhrystone 数であり，数値が大きいほどそのコンパイラが高性能であることを意味する．

この結果から，同時コピー中間コードを用いない生存区間分割では十分な性能向上が望めないことがわかる．閾値 0〜5 の場合と比べて閾値 6 の場合では若干の性能向上が見られるが，これは，干渉度が高い区間でのみ分割を行うことで，干渉やコピー命令が無意味に増加することを防げたためだと思われる．また，本手法を実装したものは COINS コンパイラと比較して明らかに性能が向上しているといえる．その性能向上率は最大で約 6%である．本研究では簡単のために制御フローにおけるループ構造に注目して分割を行うという生存区間分割アルゴリズムを用いたが，より効果の高い分割アルゴリズムに対して同時コピー中間コードを利用することで，更なる性能向上が望める．

5. おわりに

本論文では，偽干渉を発生させない生存区間分割手法を提案した．従来の生存区間分割手法では偽干渉が発生し，それによって最適なレジスタ割付けが妨げられることがあった．本手法では，同時コピー中間コードを用いることで偽干渉の発生を回避し，最適なレジスタ割付けを可能にした．

参考文献

1) Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., and Markstein, P.W.: Register allocation via coloring. *Computer Languages*, Vol.6, No.1, pp.47-57 (1981).
2) Briggs, P.: Register Allocation via Graph Coloring. *PhD thesis, Rice University* (1992).
3) Kolte, P., and Harrold, M.J.: Load/Store Range Analysis for Global Register Allocation. *In Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pp.268-277 (1993).
4) Nakaike, T., Inagaki, T., Komatsu, H., and Nakatani, T.: Profile-based global live-range splitting. *In Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pp.216-227 (2006).
5) Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, M.N.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, Vol.13, No.4, pp.451-490 (1991).
6) Park, J., and Moon, S.: Optimistic Register Coalescing. *ACM Transactions on Programming Languages and Systems.* Vol.26, No.4, pp.735-765 (2004).
7) George, L., and Appel, A.W.: Iterated register coalescing. *ACM Transactions on Programming Languages and Systems.* Vol.18, No.3, pp.300-324 (1996).
8) COINS project. http://www.coins-project.org/.