情報処理学会論文誌　プログラミング　Vol. 2　No. 2　70–79 (Mar. 2009)

並列コピーの導入による生存区間分割手法の性能向上

中林　淳一郎[†1]　片岡　正樹[†2]　古関　聰[†2]
小松　秀昭[†2]　深澤　良彰[†3]

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．そこで，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，生存区間分割には副作用が存在し，それがレジスタ割付けに悪影響を及ぼしてしまう場合があった．その副作用とは，生存区間分割によって発生する逐次コピー処理が原因となり，分割前には存在しなかった干渉が新たに発生してしまうというものである．この新たに発生する干渉を偽干渉と呼ぶ．本論文では，偽干渉の発生を回避することで生存区間分割の効果をより高める手法を提案する．偽干渉の発生を回避するためには，複数のコピー命令が並列に実行されることを表現する必要がある．そこで，我々は並列コピー中間コードを新たに定義し，逐次コピー処理の代わりにそれを利用することにした．ただし，実際に並列コピーを行うことはできないので，レジスタ割付け完了後に，並列コピーをした場合と同じ結果が得られるようなマシンコードに変換しなければならない．この変換の際には，特にコピー命令間の依存関係に注意する必要がある．以上のような手法を用いることで，生存区間分割の効果をより高めることができる．SPEC ベンチマークにおいて，本手法を実装したコンパイラが生成したコードは，本手法未実装のコンパイラが生成したコードよりも平均で約 0.9%高速であった．

Improvement of Live-range Splitting Method by Using Concurrent-copy

Junichiro Nakabayashi,[†1] Masaki Kataoka,[†2]
Akira Koseki,[†2] Hideaki Komatsu[†2]
and Yoshiaki Fukazawa[†3]

Graph coloring, a typical technique of register allocation, is prone to generate redundant spill code. Some techniques of live-range splitting have been proposed to solve this problem. However, these techniques have the side-effect which interferes with register allocation, because the interference among variables can be increased by the sequence of copy instructions which are generated by live-range splitting. The increased interference is called pseudo-interference. In this paper, we propose a new technique that can improve the performance of live-range splitting by eliminating pseudo-interference. The representation which means some copy instructions are concurrently executed is required for eliminating pseudo-interference. Therefore, we defined concurrent-copy instruction in intermediate representation and use it instead of the sequence of copy instructions. However, we must convert it into the sequence of machine codes that can obtain the same result as it after register allocation, because it can not be executed. The dependences which exist among the copy instructions must be paid attention at this conversion. In this way, the performance of live-range splitting can be improved. The codes generated by a compiler with our technique run an average of about 0.9% faster than the codes generated by a compiler without our technique in the SPEC benchmark.

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)] があげられる．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．たとえば，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．

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)] という手法が有効である．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り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．

しかし，生存区間分割には副作用が存在し，それがレジスタ割付けに悪影響を及ぼしてしまう場合があった．その副作用とは，生存区間分割によって発生する逐次コピー処理が原因となり，分割前には存在しなかった干渉が新たに発生してしまうというものである．本論文では，この新たに発生する干渉を偽干渉と呼ぶ．偽干渉が発生することで変数の干渉度が高

†1 早稲田大学大学院基幹理工学研究科
Graduate School of Fundamental 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.
†3 早稲田大学理工学術院
Faculty of Science and Engineering, Waseda University

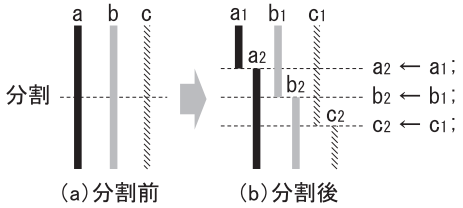


図 1　生存区間分割の例
Fig. 1　Example of live-range splitting.

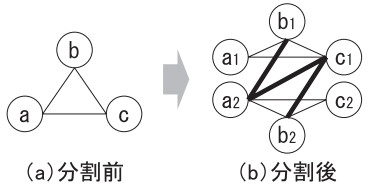


図 2　偽干渉の例
Fig. 2　Example of pseudo-interference.

まり，最適なレジスタ割付けを行えない場合がある．たとえば，生存区間分割を行う前の状態では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．

生存区間分割を行った際に偽干渉が発生する例を**図 1** に模式化する．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．生存区間分割を行うと，その分割点にコピー命令が挿入される．分割された小区間はそれぞれ別の変数として扱われるが，それらはすべて同じ値を保持しなければならないからである．そして，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．ただし，複数のコピー命令を並列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，逐次コピー処理が発生する．たとえば，図 1 では a，b，c という 3 つの変数の生存区間を分割している．そのため，分割後の状態では各変数に関するコピー命令が挿入されており，逐次コピー処理となっている．そして，この逐次コピー処理が原因となって偽干渉が発生する．たとえば，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，逐次コピー処理のために，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，生存区間分割を行う前には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並列に実行できないために逐次コピー処理が発生し，その結果，偽干渉が発生してしまう．つまり，複数のコピー命令を並列に実行できると仮定したうえ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並列に実行することを表す並列コピー中間コードを用意し，それを利用することで生存区間分割の副作用を除去する手法を提案する．これにより，生存区間分割の効果を高めることができる．また，本手法を組み込んだ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．たとえば，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 らは，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，分割を行うことで発生した冗長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副作用があるため，冗長な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 つの副作用が存在する．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逐次コピー処理が挿入され，それによって偽干渉が発生する．コアレシングではこれを回避することができない．なぜなら，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．

たとえば Park らは，積極的なコアレシングを行い，コアレシングによって統合された生存区間がスピル候補として選択された場合は，それらを統合前の状態に戻すというコアレシング手法を提案している[6)]．生存区間分割に Park らの手法を組み合わせた場合，分割された生存区間はすべてコアレシングされ，1 つの生存区間に戻った状態で彩色が行われることになる．この場合，当然偽干渉は発生しない．しかし，そもそも生存区間が分割された状態で彩色を行うことが重要であるため，コアレシングした状態で彩色を行ったのでは最適なレジスタ割付けを行えないことがある．**図 3** はその例である．図中の数値は各生存区間のスピルコストを表している．ここで，使用可能なレジスタが 1 つしかない状況であると仮定する．Park らのコアレシング手法を適用した場合は (a) の状態で彩色が行われることになる．変数 a のスピルコストは合計 11，変数 b のスピルコストは合計 15 であるため，a がスピル候補として選択され，最終的に a_1 と a_2 がスピルされることになる．しかし，(b) の状態で彩色を行えば a_2 と b_1 をスピルすることができ，これがスピルコストを最も小さくすることができる最適なレジスタ割付けである．このように，コアレシングでは生存区間分割の効果を維持したまま偽干渉の発生を防ぐことはできないため，それに代わる手法が必要である．

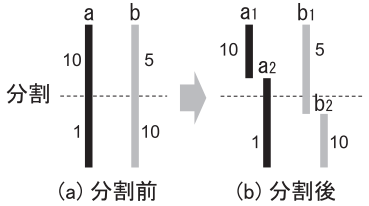


図 3　生存区間分割の効果
Fig. 3　Effect of live-range splitting.

本研究と似た手法をとっている研究として，Hack らによる SSA 形式におけるレジスタ割付けに関する研究があげられる[8)]．連続した複数の ϕ 関数をそれぞれコピー命令に変換する場合，挿入されるコピー命令によって新たな干渉が発生してしまうことがある．Hack らはこれを回避するために，連続した ϕ 関数を 1 度に実行されるコピー命令の集合として考え，最終的にそれをスワップ操作に変換するという手法をとっている．この点が，後述する本手法の流れと似ている部分である．

このように，Hack らの研究と本研究は問題を解決する手段はよく似ているが，解決する問題は異なっている．Hack らは，彩色が完了した後の ϕ 関数を除去する段階で発生する新たな干渉によって，彩色の結果がそこなわれることを防ぐために前述の手法を用いている．対して，本研究が注目している生存区間分割とは，彩色を行う前に生存区間を分割することで，彩色の結果を向上させるというものである．そして，本研究は偽干渉の発生を防ぐことで生存区間分割手法の性能をより高めるために，並列コピー中間コードを導入する．すなわち，彩色の結果を維持することを目的としているのか，あるいは，彩色の結果を向上させることを目的としているのか，という点が Hack らの研究と本研究の違いである．

3. 本手法の特徴

前章で述べたように，生存区間分割には偽干渉を発生させてしまうという副作用があり，それがレジスタ割付けの結果に悪影響を与えてしまうという問題点があった．そこで，我々は偽干渉の発生を回避する手法を提案する．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並列に実行できればよい．コピー命令を並列に実行できると仮定した場合の生存区間分割の例を**図 4** に模式化する．図 2 の (b) と比較することで，図 4 の状況では偽干渉が発生していないことが分かる．

そこで，我々は複数のコピー命令を並列に実行することを表す並列コピー中間コードを新たに定義した．そして，生存区間分割によって発生する逐次コピー処理をそれに置き換えることで，複数のコピー命令が並列に実行されることを表現できるようにした．このようにして，コピー命令の並列実行が可能であると仮定したうえでレジスタ割付けを行えば，偽干渉

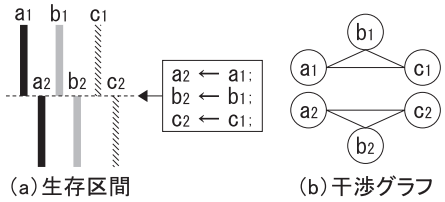


図 4　並列コピーを用いた生存区間分割の例
Fig. 4　Example of live-range splitting by concurrent-copy.

が発生することはない．しかし，現在のプロセッサでは複数のコピー命令を 1 サイクルで並列して行うことはできない．そのため，レジスタ割付けの完了後に，並列コピー中間コードをプロセッサが処理できる形に変換する必要がある．その際には，並列コピーをした場合と同じ結果が得られるようなコードに変換しなければならない．

本手法の最大の特徴は，あらゆる生存区間分割手法に適用することができるという点である．本手法は分割の仕方には依存していないため，どのような分割アルゴリズムを用いていても偽干渉の発生を防ぐことができる．特に，制御フローのエッジにおいて分割を行う手法に対して適用した場合に，高い効果が得られると考えられる．そのような手法は逐次コピー処理が発生しやすく，偽干渉が多く発生するからである．

4. 本手法の概要

本手法の流れを**図 5** に示す．以下では，各ステップの概要を説明していく．

4.1 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．前述のように，分割のアルゴリズムはどのようなものでもよい．本手法はあらゆる生存区間分割手法に対して適用できる．

4.2 並列コピー中間コードへの置き換え

生存区間分割を行った後に，分割によって発生した逐次コピー処理を並列コピー中間コードで置き換える．その例を**図 6** に示す．図のように，逐次コピー処理となっていた複数のコピー命令は，1 つの並列コピー中間コードにまとめられる．そして，それらのコピー命令は並列に実行されると仮定してレジスタ割付けを行う．

4.3 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．ただし，生存区間解析および干渉グラフ生成の際には，並列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，並列コピー中間コードに含まれているコピー命令は並列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．それらのコピー命令の左辺変数の生存区間は並列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，並列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，偽干渉は発生しないことになる．

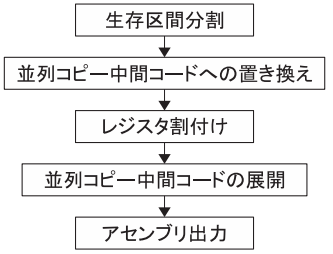


図 5　本手法の流れ
Fig. 5　Flow of our method.

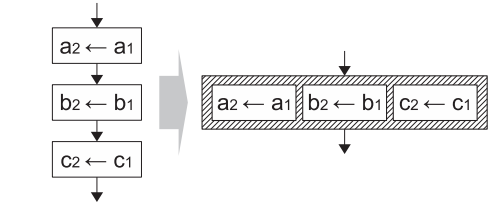


図 6　並列コピー中間コードの例
Fig. 6　Example of concurrent-copy instruction.

4.4 並列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．ただし，アセンブリコードを出力する際には並列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並列に実行することは不可能であり，並列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並列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，並列コピー中間コードがあった地点に挿入する必要がある．本論文では，この処理を並列コピー中間コードの展開と呼ぶ．

並列コピー中間コードを展開する際には，特にコピー命令を並べる順番に注意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並列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である．

展開処理は大きく 2 段階に分けられる．まず，スピルされた変数に関するコピー命令を並列コピー中間コードから取り出し，その前後に挿入する．次に，残りのコピー命令をすべて取り出してコピー命令間の依存関係に基づいて並べ，並列コピー中間コードと置き換える．以下では，これらの処理の詳細を説明していく．

4.4.1　スピルされた変数に関するコピー命令の展開

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，それらを並列コピー中間コードから抜き出して次のように挿入する．左辺変数がスピルされているコピー命令は，並列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．並列コピー中間コードよりも後に実行されるようにすると，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並列コピー中間コードの前に挿入する．逆に，右辺変数がスピルされているコピー命令は，並列コピー中間コードの後に挿入する必要がある．

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．

このとき，左辺変数と右辺変数がともにスピルされているコピー命令は，並列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 7** に示す．図 7 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ともにスピルされているコピー命令である．

4.4.2　レジスタが割り付けられた変数に関するコピー命令の展開

スピルされた変数に関するコピー命令の展開を行うことで，並列コピー中間コードに残っているコピー命令は，両辺の変数にレジスタが割り付けられたもの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以下のような手順で順番に並べ，並列コピー中間

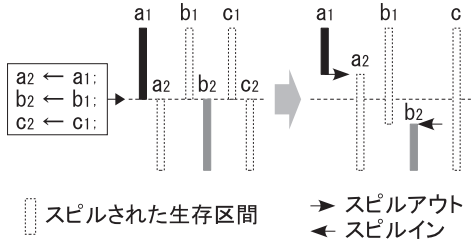


図 7　スピルされた変数に関するコピー命令の展開
Fig. 7　Expansion of concurrent-copy instruction.

コードと置き換えることで展開処理は完了する．

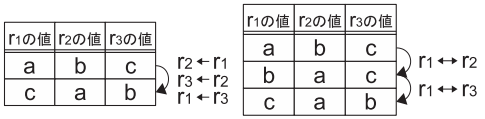
まず，左辺変数と右辺変数に同じレジスタが割り付けられたコピー命令を並列コピー中間コードから削除する．そのようなコピー命令はあるレジスタから同じレジスタにその値をコピーするという無意味な処理であるため，削除してしまっても問題がない．

次に，残りのコピー命令，すなわち左辺変数と右辺変数に異なるレジスタが割り付けられたコピー命令を並列コピー中間コードから抜き出して並べる．ただし，これらのコピー命令の間には依存関係が存在する場合があり，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．たとえば，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 つのコピー命令が並列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並列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．たとえば，以下のような 3 つのコピー命令があるとする．

(1)　$r_2 \leftarrow r_1$
(2)　$r_3 \leftarrow r_2$
(3)　$r_1 \leftarrow r_3$

(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



(a) 並列コピーを行った場合　(b) スワップを利用した場合

図 8　スワップ操作の利用例
Fig. 8　Example of swap operation.

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 8** に示す．図 8 の (a) は，3 つのコピー命令を並列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，コピー命令を並列に実行した場合と同じ状態を再現できることが分かる．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．

以上のように変換されたコピー命令列を並列コピー中間コードと置き換えることで，展開処理が完了する．

5. 実験と結果

5.1 実 験 内 容

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，SPEC ベンチマークを用いて性能の比較実験を行った．

(1)　COINS コンパイラ[9]（Ver. 1.4.3.3）
(2)　生存区間分割手法を組み込んだ COINS コンパイラ
(3)　生存区間分割手法と本手法を組み込んだ 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，これをベースに比較実験を行うことにした．

(2) と (3) のコンパイラの生存区間分割アルゴリズムは，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．生存区間分割手法については前述のようにいくつかの手法が提案されているが，今回は簡単のために，この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ため，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に分割することができると考えられる．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と考えられる．

基本的な生存区間分割アルゴリズムは上記のとおりであるが，さらに以下の 2 点の分割条件を定め，アルゴリズムが若干異なるものを複数用意して比較することにした．1 つ目の条件は，すべてのループ構造で分割を行うのか，最内ループでのみ分割を行うのかというものである．2 つ目の条件は，ループ構造内の干渉度が x 以上であれば分割を行い，x 未満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「分割干渉度 x」という形式で示す．たとえば，分割干渉度 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．今回は分割干渉度を 4〜6 の間で変化させた．これら 2 つの条件を組み合わせ，合計 6 パターンの分割アルゴリズムを用意した．

なお，オリジナルの COINS には標準でコアレシングによる最適化が組み込まれているが，今回はその処理を削除した状態で実験を行った．本手法では 1 つの並列コピー中間コードで複数の変数が定義されるが，COINS のコアレシングではこのようなことは想定されていない．そのため，並列コピー中間コードに出現する変数の順番が変わるだけで，コアレシングの結果が異なるものになってしまうといった問題がある．これを回避するためには，1 つの命令で複数の変数が定義されることを想定した形に COINS のコアレシング処理を拡張する必要があるが，現状ではそれが完了していないため，今回はコアレシング処理を削除することで対応した．オリジナル COINS とコアレシング処理を削除した COINS を用意し，SPEC CINT 2000 のいくつかのプログラムをコンパイルおよび実行した際の実行速度比率を，オリジナル COINS を 100%として**図 9** に示す．図から分かるように，コアレシングを削除したことで COINS の性能が大きく低下することはなかった．よって，コアレシングを削除した状態でも本手法の効果を確かめることに影響はないと考えられる．

実験の内容は，上記の各コンパイラで SPEC CINT 2000 のいくつかのプログラムをコンパイルし，その実行時間を比較するというものである．実行環境は，CPU が Pentium4 (3.4 GHz)，メモリ 2 GB，OS は WindowsXP Professional である．

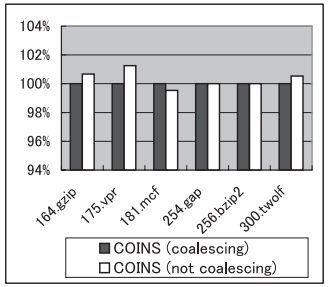


図 9　COINS におけるコアレシングの効果
Fig. 9　Effect of coalescing in COINS.

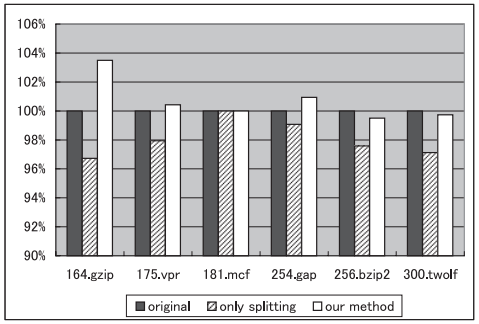


図 10　実行速度比率（分割条件：全ループ，分割干渉度 4）
Fig. 10　Ratio of execution speed (splitting condition: all loop, register pressure is 4 or more).

5.2 実験結果

実験の結果を**図 10**，**図 11**，**図 12**，**図 13**，**図 14**，**図 15** に示す．各図は，オリジナル COINS を 100%とした場合のプログラム実行速度の比率を表している．すなわち，グラフが長いものの方がプログラムをより高速に実行できたということであり，高性能なコンパイラであるといえる．また，実行速度比率の相乗平均を**表 1** に示す．

相乗平均した結果を比較すると，全ループかつ分割干渉度 5 で分割を行った場合が最も性能が高く，オリジナル COINS と比べて約 0.9%向上という結果であった．これは，プログラムが干渉グラフの彩色に利用できるレジスタの数 N が関係していると考えられる．x86 系のプロセッサにおいて，干渉グラフの彩色に利用できる汎用レジスタは 4 つ，すなわち，$N = 4$ である．そのため，干渉度 4 以下であれば干渉している変数をすべてレジスタに配置することができ，スピルが発生することは少ない．よって，干渉度 4 のループで分割を行っても，冗長なコピー命令を発生させるだけになってしまう場合がある．対して，干渉度が 5 以上の場合はレジスタが不足してスピルが発生することが多い．つまり，干渉度が 5 以上のループで分割を行えば，生存区間分割および本手法によるスピルの最適化が効果を発揮できる．分割干渉度 6 の場合は，干渉度が 5 であるループでは分割を行わないため，分割干渉度 5 の場合よりも性能が低いと考えられる．このことから，分割干渉度が $N + 1$ の場合に最も性能が高くなると考えられる．

また，最内ループでのみ分割を行った場合は，すべてオリジナル COINS よりも性能が低下するという結果になった．これは，多重ループに対して分割を行った場合に，外側ループ

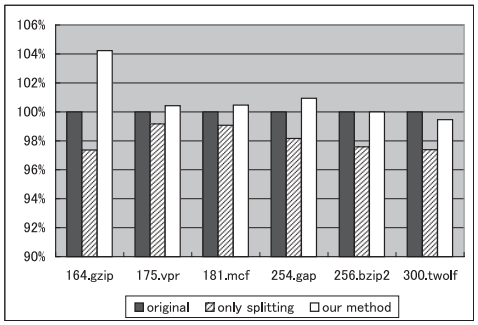


図 11　実行速度比率（分割条件：全ループ，分割干渉度 5）
Fig. 11　Ratio of execution speed (splitting condition: all loop, register pressure is 5 or more).

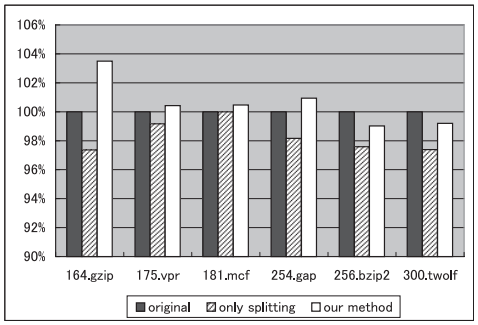


図 12　実行速度比率（分割条件：全ループ，分割干渉度 6）
Fig. 12　Ratio of execution speed (splitting condition: all loop, register pressure is 6 or more).

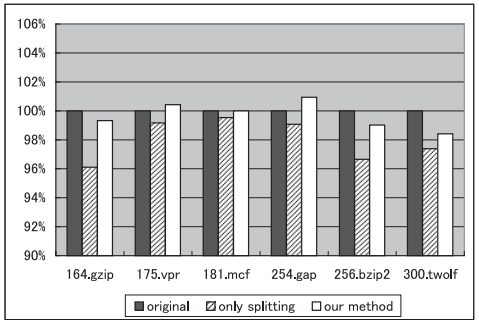


図 13　実行速度比率（分割条件：最内ループ，分割干渉度 4）
Fig. 13　Ratio of execution speed (splitting condition: inner loop, register pressure is 4 or more).

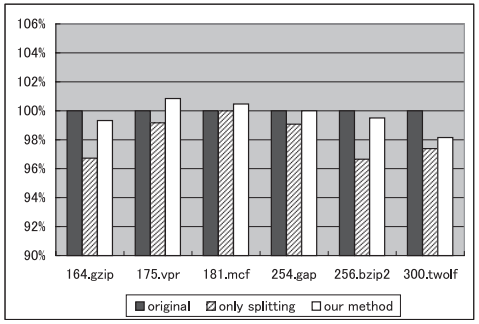


図 14　実行速度比率（分割条件：最内ループ，分割干渉度 5）
Fig. 14　Ratio of execution speed (splitting condition: inner loop, register pressure is 5 or more).

内部のコピー命令が増加してしまったことが原因であると考えられる．最内ループの内部で使用される変数に対して分割を行う際に最内ループでのみ分割を行うと，最内ループの前後，すなわち，外側ループの内部にコピー命令が残ってしまう可能性が高い．対して，すべてのループで分割を行う場合は外側ループの前後でも分割が行われるため，外側ループの前後に挿入されたコピー命令が最内ループの前後に挿入されたコピー命令の役割を果たし，最内ループ前後のコピー命令を削除できる場合がある．よって，外側ループ内部にコピー命令が残ってしまう可能性が低くなるので，最内ループでのみ分割を行う場合に比べてコピー命令の実行回数が少なくなり，性能が向上したと考えられる．

以上のことから，全ループかつ分割干渉度が $N+1$ である生存区間分割手法に本手法を適用したものが最も性能が高くなるということが今回の実験で確認できた．

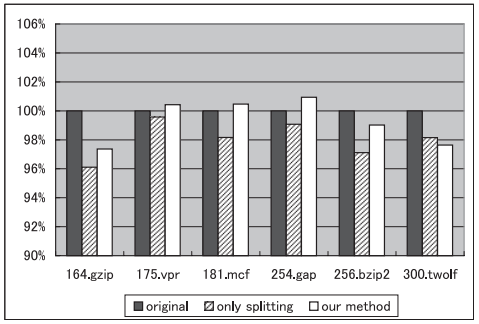


図 15　実行速度比率（分割条件：最内ループ，分割干渉度 6）
Fig. 15　Ratio of execution speed (splitting condition: inner loop, register pressure is 6 or more).

表 1　実行速度比率の相乗平均
Table 1　Geometric average of the ratio of execution speed.

	生存区間分割のみ	本手法
全ループ，分割干渉度 4	98.1%	100.7%
全ループ，分割干渉度 5	98.1%	100.9%
全ループ，分割干渉度 6	98.3%	100.6%
最内ループ，分割干渉度 4	98.0%	99.7%
最内ループ，分割干渉度 5	98.2%	99.7%
最内ループ，分割干渉度 6	98.0%	99.3%

表 2　生成されたスワップ命令の数
Table 2　Number of SWAP operations.

	164.gzip	175.vpr	181.mcf	254.gap	256.bzip2	300.twolf
全ループ，分割干渉度 4	22	8	0	200	6	39
全ループ，分割干渉度 5	21	9	0	194	6	56
全ループ，分割干渉度 6	21	13	0	174	3	27
最内ループ，分割干渉度 4	8	5	0	138	2	66
最内ループ，分割干渉度 5	7	5	0	124	2	21
最内ループ，分割干渉度 6	7	5	0	98	2	16

ちなみに，この実験において本手法が生成したスワップ命令の数は**表 2** のようになった．スワップ命令は主に，関数からの返り値を格納する変数やループ変数等，割り付けられるレジスタが限定されている変数が存在する場合に必要になることがある．また，それ以外に

も彩色の仕方によっては冗長なスワップ命令が生成されてしまう場合があると考えられる．本手法は彩色の方法にはまったく関与しないため，ある変数に割付け可能なレジスタが複数ある場合，そのうちのどのレジスタがその変数に割り付けられるかについてはCOINSの実装に依存している．そのため，分割された2つの変数には可能な限り同じレジスタを割り付けるように彩色方法を変更することで，スワップ命令の数を少なくし，性能をさらに向上させることができるのではないかと考えている．

6. おわりに

本論文では，生存区間分割手法の効果をより高める手法を提案した．生存区間分割手法には偽干渉を発生させてしまうという副作用があり，それがレジスタ割付けに悪影響を与えていた．本手法では，並列コピーを導入す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．実験の結果，COINSコンパイラと比較して平均で約0.9%の性能向上がみられた．より高性能な生存区間分割手法と組み合わせたり，本手法を考慮した最適化に変更したりすることで，さらなる性能の向上が図れるのではないかと考えている．

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, Ph.D. Thesis, Rice University (1992).
3) Kolte, P. and Harrold, M.J.: Load/Store Range Analysis for Global Register Allocation, *Proc. 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, *Proc. 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 Trans. Prog. Lang. Syst.*, Vol.13, No.4, pp.451–490 (1991).
6) Park, J. and Moon, S.: Optimistic Register Coalescing, *ACM Trans. Prog. Lang. Syst.*, Vol.26, No.4, pp.735–765 (2004).
7) George, L. and Appel, A.W.: Iterated Register Coalescing, *ACM Trans. Prog. Lang. Syst.*, Vol.18, No.3, pp.300–324 (1996).
8) Hack, S., Grund, D. and Goos, G.: Register Allocation for Programs in SSA-Form, *Lecture Notes in Computer Science*, Vol.3923, pp.247–262 (2006).
9) 中田育男，渡辺　坦，佐々政孝，森公一郎，阿部正佳：COINSコンパイラ・インフラストラクチャの開発，コンピュータソフトウェア，Vol.25, No.1, pp.2–18 (2008).

(平成20年 9 月28日受付)
(平成20年12月26日採録)

中林淳一郎

1984年生．2007年早稲田大学理工学部コンピュータ・ネットワーク工学科卒業．現在，同大学大学院基幹理工学研究科情報理工学専攻修士課程に在学中．

片岡　正樹

1980年生．2008年早稲田大学大学院理工学研究科コンピュータ・ネットワーク工学専攻博士課程修了．同年日本IBM（株）東京基礎研究所入社．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．