スライドウィンドウを考慮したレジスタ割付

秡川友宏[†] 添野元秀[†]
山下義行[††] 中田育男[†††]

主記憶アクセスのレイテンシを隠蔽する手法として，ソフトウェア・パイプライニングが注目されている．ソフトウェア・パイプラインコードを自然に表現でき，かつ高速に実行できるアーキテクチャとしてはスライドウィンドウ・アーキテクチャがあるが，スライドウィンドウ・アーキテクチャに対し従来のレジスタ割付法をそのまま適用することは困難である．本論文では，スライドウィンドウを考慮したレジスタ割付法として，(A) スライドレジスタ干渉グラフおよびSlide Coloring Algorithmからなるフレームワーク，(B) Spiral GraphおよびShort Bridge Algorithmからなるフレームワークを提案し，それらによるレジスタ割付結果を報告する．Livermore Fortran Kernelによるレジスタ割付実験においては両者ともに9割以上の例題に対して最適解を与えた．さらに，自動生成した10,000例のループプログラムを用いて両者の傾向を調べたところ，フレームワーク (B) の方がより効果的なレジスタ割付を行っていることが分かった．

Register Allocation Frameworks for Slide-Window Architecture

TOMOHIRO HARAIKAWA,[†] MOTOHIDE SOENO,[†]
YOSHIIYUKI YAMASHITA[††] and IKUO NAKATA[†††]

Software-pipelining is widely applied as an effective method to hide main memory access latency. Using Slide-Window Architecture, pipelined code can be represented straightly and executed efficiently. However, it is difficult to apply the conventional framework of register allocation as is to this architecture. This paper proposes following two register allocation method for Slide-Window Architecture: (A) a framework consisting of *Slide-Register Interference Graph* and *Slide Coloring Algorithm*, (B) a framework consisting of *Spiral Graph* and *Short Bridge Algorithm*, and reports their experimental results. In the register allocation experiment using Livermore Fortran Kernel, both frameworks gave the optimal solution to more than 90% of examples. Furthermore, two methods were investigated using 10,000 loop programs that were generated automatically. The result indicates that the framework (B) is more effective.

1. はじめに

CP-PACS（SR-2201）は，1996年にTOP500 Listで世界最速と認定された超並列計算機である☆．CP-PACSは，ハイパクロスバ・ネットワーク[1]で結合されたピーク性能300MFLOPSのノードプロセッサ2,048台からなり，全体として600GFLOPSのピーク性能を持つ．スライドウィンドウ・アーキテクチャは，命令レイテンシ—特に主記憶アクセスレイテンシ—の隠蔽を容易にし，個々のプロセッサの高速動作を支援するための特殊なハードウェア機構である．

演算速度に対して主記憶アクセス速度が低速なシステムでは，キャッシュメモリを用いてこのレイテンシを回避するのが主流であるが，CP-PACSが対象としている大規模科学技術計算においては，ループを用いて広範にデータをスキャンするため主記憶の参照が局所性に乏しく，高レートのキャッシュヒットを仮定できない．

このような状況下で主記憶アクセスのレイテンシを回避する方法としては，ソフトウェア・パイプライニングを用い，先行ステージでロード命令を発行するのが有効である．スライドウィンドウ・アーキテクチャはレジスタウィンドウの外側のレジスタに対してロードを行う機能と，後からウィンドウを移動させる機能を提供するこ

† 筑波大学工学研究科
Doctoral Program in Engineering, University of Tsukuba
†† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba
††† 図書館情報大学図書館情報学部
University of Library and Information Science

☆ 1997年11月現在第4位.

とによって，使用可能レジスタ数を圧迫せずに先行ロードを実現できる[2]．

スライドウィンドウ・アーキテクチャにはまた，命令スケジューリングもごく自然な発想に基づいて素直に実現できるという利点がある[3]．ソフトウェア・パイプライン化されたループのカーネル部における，繰返し1回あたりのマシンサイクル数を**イテレーション開始間隔**または**イテレーション立ち上げ間隔**（*II*: Initiation Interval）[4]という☆．通常のアーキテクチャにおいては，繰返し1回以上にわたって同じレジスタに値を保持し続けることはできないから，*II* がそのループに含まれる命令の最大レイテンシを下回らないようにスケジューリングがなされなければならない．ところが，スライドウィンドウ・アーキテクチャにおいては，ウィンドウを移動させるというレジスタリネーミングの効果で論理レジスタ番号が変化するため，同じ物理レジスタに繰返し1回より長い時間，値を保持することができる．このため，最大レイテンシに制約されずに自由に *II* を決定できるのである．

このような特徴を持つスライドウィンドウ・アーキテクチャにおいては，レジスタリネーミングの副作用により従来のレジスタ割付法が単純には適用できないと考えられてきた[3]．本論文では，分岐を含まないループプログラムを対象に，スライドウィンドウを考慮したレジスタ割付法を提案する．ループプログラムのレジスタ割付法としては，(a) レジスタ干渉グラフに基づくフレームワーク[5)~7)]，(b) Cyclic Interval Graph に基づくフレームワーク[8]の2つが広く知られている．筆者らは，これらをそれぞれ拡張し，(A) スライドレジスタ干渉グラフに基づくフレームワーク，(B) Spiral Graph に基づくフレームワークとして提案する．また，実験によってその性能評価を行う．

2. 基本事項

2.1 スライドウィンドウ・アーキテクチャ

スライドウィンドウ・アーキテクチャは，主として主記憶アクセスのレイテンシを隠蔽するために提案され[2]，超並列計算機CP-PACSのノードプロセッサHARP-1Eに採用された．HARP-1E は PA-RISC1.1[9] アーキテクチャの浮動小数レジスタセットに**スライドウィンドウ機構**を付加したものである（**図1**）[10]．

HARP-1E のレジスタはグローバルレジスタとローカルレジスタからなる．グローバルレジスタの個数 g は

☆ 1マシンサイクルごとに次の命令の実行を開始するプロセッサ（ノンブロッキング型のプロセッサという）においては，*II* は単に，繰返し1回あたりの命令ステップ数と考えてよい．

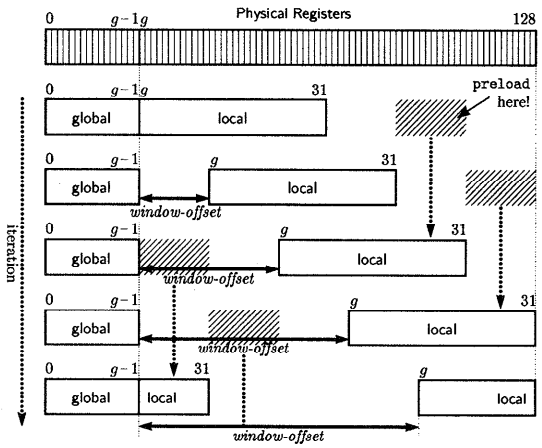


図1 スライドウィンドウ・アーキテクチャ
Fig. 1 Slide-Window Architecture.

4，8，12から選択可能であり，32から g を減じた個数がローカルレジスタの個数となる．グローバルレジスタは従来のアーキテクチャにおけるレジスタと同様であり，従来のレジスタ割付フレームワークを適用できる．一方，ローカルレジスタは，レジスタウィンドウのオフセットを変更することによっていっせいにリネーム可能であるという特徴を持つ．HARP-1E はこのための命令 **FWSTPset**（set Floating Window STart Pointer）命令を持つ．また，**FRPreload** および **FRPoststore** という命令を持ち，ウィンドウの外側のレジスタに対してもウィンドウオフセットからの相対指定でロード/ストアを行うことができる．

このような機構を用いると，ループ構造のプログラムに対して主記憶レイテンシを隠蔽する目的コードを生成することができる．現在の繰返しにおいては，すでにレジスタにロードされた値を用いて演算を行うとともに，数回先の繰返しで必要となる値を **FRPreload** 命令を用いて将来ウィンドウが移動する位置（図1斜線部）にロードしておくのである．

FWSTPset 命令はジャンプ命令と同時実行可能であるため，多くの場合はループ末尾にジャンプ命令とともに置かれる．各繰返しの末尾においていくつウィンドウをスライドさせるかは，繰返し1回あたりいくつの値をメモリからロードするかによって決めている．このことについては5章でやや詳しく述べる．

本論文では，ローカルレジスタを，その意味から**スライドレジスタ**とよぶことにする．図1のように，スライドレジスタのレジスタ番号は g から始まるのであるが，本論文では0から始まるものとし，**SR0, SR1,...** と書く．また，簡単のため，レジスタ番号を K 移動させるための仮想命令 slide K（**FWSTPset** $-K$ に相当）を

導入し，**FRPreload** 命令および **FRPoststore** 命令は単に preload，poststore と書く．

2.2 基本方針

コード生成には，大別して，命令スケジューリングを行ってからレジスタ割付を行う方法と，レジスタ割付を行ってからスケジューリングを行う方法がある[4]が，我々は前者を採用している．これは，スケジューリングが素直に行えるというスライドウィンドウの特徴を重視していることと，スケジューリングが定まらないとレジスタのリネーミング情報が定まらず，レジスタ割付を先に行うのに困難がともなうことによる．

本論文では，スケジューラから与えられた II やスライド量を含む情報をもとに占有レジスタ数を最小化することを考え，占有レジスタ数が最小になるとき，その割付が最適であるということにする．これは，占有レジスタ数が物理レジスタ数以下であるときに最適とする条件に比べて厳しく，奇妙に思えるかもしれない．我々がこのように定めているのは，効果的なレジスタ割付の結果レジスタに十分な空きができれば，将来的にこの情報をスケジューラにフィードバックすることにより，ループ・アンローリングなどの技法を適用してさらに高速なコードを生成できる可能性があるためである．

以下，従来アーキテクチャにおけるレジスタ割付法を引用し，それと対比させながらスライドウィンドウ・アーキテクチャにおけるレジスタ割付法を提案する．

3. スライドレジスタ干渉グラフによるフレームワーク

レジスタ割付法は，一般にグラフとアルゴリズムからなる．**レジスタ干渉グラフ**（Register Interference Graph）と**彩色アルゴリズム**によるフレームワークは，現在まで最もよく利用されてきた手法である[5)~7)]．本章では，スライドウィンドウを考慮したレジスタ割付法として，従来のグラフとアルゴリズムを拡張し，**スライドレジスタ干渉グラフ**と **Slide Coloring Algorithm** からなるフレームワークを提案する．

3.1 従来手法 — レジスタ干渉グラフ

レジスタ干渉グラフは，各々の変数もしくは一時名（以下，単に変数という）を頂点とし，2つの変数の生存期間が重なるときに，その2つの頂点を無向線分で結んだグラフである（**図2**）．各頂点から出ている線分の数をその頂点の度数という（図2ではイタリック数字で示した）．

彩色アルゴリズムは，レジスタ番号を色に見立て，レジスタ割付問題をグラフの彩色問題として解こうというものである．生存期間の重なる変数どうしに同じレジスタ番号が割り当てられることはないから，無向線分で結ばれた頂点どうしは別な色で彩色されなければならない．以下，説明を簡潔に記述するため，この条件を満たさない彩色を，単に**矛盾する**ということにする．

```
do {
0:  load   v1
1:  add    v6, v1 -> v4
2:  mult   v4, v4 -> v5
3:  add    v5, v1 -> v2
4:  mult   v1, v2 -> v3
5:  add    v2, v3 -> v6
6:  store  v2
}
```

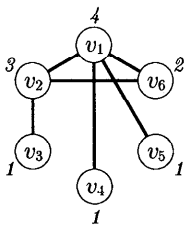


図2 Register Interference Graph
Fig. 2 Register Interference Graph.

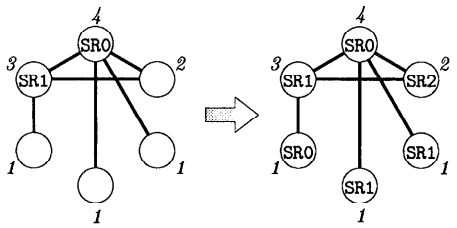


図3 COLORING によるレジスタ割付のようす
Fig. 3 Register allocation via COLORING.

この種の彩色問題では，制約の大きいものから先に彩色し，制約の少ないものを最後に彩色するのが一般的である．なるべく少ないレジスタ数で彩色するためには，ごく簡単には，次のような近似アルゴリズムで実現できる．

アルゴリズム1（COLORING）

(1) 未彩色の頂点のうち，度数の最も大きいものを1つ選ぶ．なければ終了．
(2) (1)で選んだ頂点と無向線分で結ばれている頂点のうち，すでにレジスタ番号が付けられているものがあれば，そのリストを作る．
(3) (1)で選んだ頂点を，リストに含まれない最小の非負整数で彩色する．
(4) (1)へ進む． ■

このアルゴリズムによってレジスタを割り付けているようすを，**図3** に示す．

3.2 提案手法 — スライドレジスタ干渉グラフ

上記のフレームワークをスライドレジスタに適用できるように拡張する．

図4 に示したプログラムは，図2のプログラムにスライド命令を付加したものである．このプログラムにおいては，v_6 は，生存期間がステップ **5** から（次の繰返しの）ステップ **1** までであるから，生存中にスライド命令を通過してレジスタ番号が変化してしまう．そのため，

```
do {
0:  load   v1
1:  add    v6, v1 -> v4
2:  mult   v4, v4 -> v5
3:  add    v5, v1 -> v2
4:  mult   v1, v2 -> v3
5:  add    v2, v3 -> v6
6:  store  v2
    slide +1
}
```

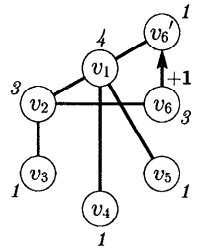


図4 Slide-Register Interference Graph
Fig. 4 Slide-Register Interference Graph.

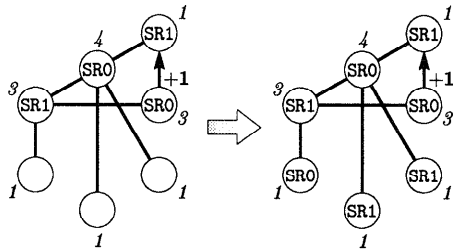


図5 SLIDECOLORING によるレジスタ割付のようす
Fig. 5 Register allocation via SLIDECOLORING.

このようなレジスタ番号の変化を表すことのできるグラフとして**スライドレジスタ干渉グラフ**（Slide-Register Interference Graph）を考える．

スライドレジスタ干渉グラフでは，元の v_6 を，スライド命令通過前後で分割して考える．具体的には通過前を v_6，通過後は通過前の名前に $'$ を付して v_6' とし，その間を有向線分で結ぶ（図4）．有向線分上にはレジスタ番号の変化量 S を記し，有向線分の元の頂点のレジスタ番号に S を加えると有向線分の先の頂点のレジスタ番号になるものと定める（図4ではボールド数字で示した）．このようにしてスライド命令による依存を表現する．

有向線分をふまえて，頂点の度数を定義する．

定義1（頂点の度数） スライドレジスタ干渉グラフにおける各頂点の度数を次のように定める．

- **その頂点から有向線分が出ていないとき**
  頂点から出ている無向線分の数
- **その頂点から有向線分が出ているとき**
  頂点から出ている有向，無向線分の数と，有向線分の先にある頂点の度数の合計 ■

アルゴリズムとしては，以下のものを用いる．

アルゴリズム2（SLIDECOLORING）

(1) 未彩色の頂点のうち，度数の最も大きいものを1つ選ぶ．なければ終了．

(2) (1)で選んだ頂点と無向線分で結ばれている頂点のうち，すでにレジスタ番号が付けられているものがあれば，そのリストを作る．

(3) (1)で選んだ頂点を，リストに含まれない最小の非負整数で仮彩色する．

(4) 現在の頂点から有向線分が出ているとき，有向線分に付された変化量を現在のレジスタ番号に加えて，有向線分の先の頂点を仮彩色する．これを有向線分のない頂点に達するまで繰り返す．

(5) 仮彩色を検証する．矛盾することが分かったなら(3)で選んだ非負整数をリストに追加し，仮彩色をすべて破棄して(3)からやり直す．矛盾しないなら仮彩色を確定して(1)へ進む． ■

このアルゴリズムによってレジスタを割り付けているようすを，**図5** に示す．

4. Spiral Graph によるフレームワーク

ループプログラムに特化したレジスタ割付法として **Cyclic Interval Graph** と **Fat Cover Algorithm** によるフレームワークが知られている[8]．本章では，スライドウィンドウを考慮したレジスタ割付法として，従来のグラフを拡張した **Spiral Graph** と，Spiral Graph のために構成した **Short Bridge Algorithm** を提案する．

4.1 従来手法 —Cyclic Interval Graph

Cyclic Interval Graph は，Fat Cover Algorithm とともにループプログラムのレジスタ割付フレームワークとして提案された[8]．Cyclic Interval Graph は，横軸にループ内の命令ステップ番号，縦軸にレジスタ番号をとった枠を用いて記述し，変数 v_i の生存期間は線分で表現する（**図6**）．

Cyclic Interval Graph では，v_1 のように繰返しをまたがない変数の生存期間を **interval** と呼び，$v_1 = [0,4)$ のように表す．最後の点そのものは含まない半開区間であることに注意されたい．また，v_6 のように繰返しをまたいで生存する変数の生存期間を **cyclic interval** と呼び，$v_6 = ([0,1),[5,6])$ のように表す．

定義2（グラフの幅 *width*） Cyclic Interval Graph G において，あるステップ s に生存している変数の数を s における G の**幅**といい，$width\,(G,s)$ と書く． ■

定義3（W_{max}，W_{min}） Cyclic Interval Graph G における幅の最大値，最小値をそれぞれ $W_{max}(G)$，$W_{min}(G)$ と書く． ■

定義4（fat cover） Cyclic Interval Graph G の変数 v に対し，次の条件をみたす変数からなる G の部分グラフ F を，v に対する G の **fat cover** という．

- 各ステップにおいて，生存している F の変数はたかだか1つである．

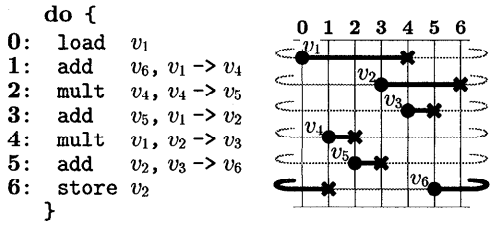


図6 ループと Cyclic Interval Graph による表現
Fig. 6 Loop representation in Cyclic Interval Graph.

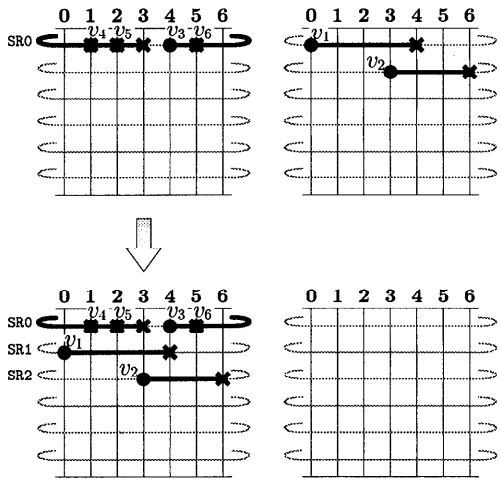


図7 Fat Cover Algorithm によるレジスタ割付のようす
Fig. 7 Register allocation via Fat Cover Algorithm.

- $width\,(G,s) = W_{max}(G)$ であるステップ s においては，F の変数が生存している． ■

4.2 Fat Cover Algorithm

Cyclic Interval Graph 上に線分で表された変数の効果的なレジスタ割付手法として提案されたアルゴリズムが Fat Cover Algorithm である[8]，

アルゴリズム 3 (Fat Cover Algorithm)

(1) Cyclic Interval Graph G において，cyclic interval の 1 つに着目する．これを v_c とする．

(2) v_c に対する fat cover を探し出し，それらを彩色して取り除く．

(3) Cyclic interval がなくなるまで (1)，(2) を繰り返す．

(4) グラフを左からサーチし，残る変数を彩色して取り除く．

(5) グラフが空になるまで (4) を繰り返す． ■

このアルゴリズムによってレジスタを割り付けているようすを，左側を彩色済の変数，右側を未彩色の変数として**図 7** に示す．

4.3 提案手法 —Spiral Graph

Spiral Graph 自体は Cyclic Interval Graph のスライドウィンドウ・アーキテクチャへの自然な拡張であるが，それと組み合わせる **Short Bridge Algorithm** は，Fat Cover Algorithm とはまったく異なるアプローチのアルゴリズムである．そこで，本節では，項を設けて詳しく説明する．

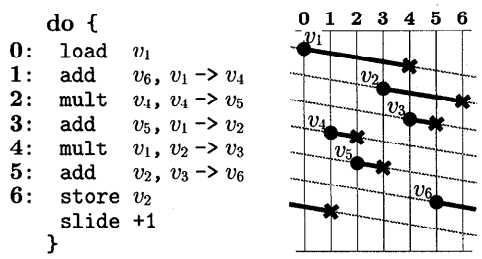


図8 Spiral Graph
Fig. 8 Spiral Graph.

4.3.1 Spiral Graph

Spiral Graph は，Cyclic Interval Graph 同様，横軸にループ内の命令ステップ番号をとったグラフを用いて表すが，Cyclic Interval Graph では水平であった横軸がスライド命令のために傾いているのが特徴である（**図 8**）．Spiral Graph 上においても，変数の生存期間は線分で表現する．ただし，Cyclic Interval Graph では $v_6 = ([0,1),[5,6])$ のように 2 区間の組で表されていた cyclic interval は，イテレーション立ち上げ間隔 II を終点に加え，interval 同様 $v_6 = [5,8)$ などと表すことにする．

Spiral Graph は，ループ 1 周の合計スライド量，すなわち，あるレジスタが次の繰返しでいくつ先の番号になるのかのみを保存したらせん状のグラフである．スライド命令は通常ループの末尾に置かれるのであるが，グラフ自体は，より一般的に複数のスライド命令を含むループも扱えるように構成しておく．複数のスライド命令を含むループを Cyclic Interval Graph 風に表現すると，横軸が途中で何度か折れ曲がったグラフ（**図 9** 最左）になる．この例ではループ 1 周の合計スライド量が 3 であるから，3 重らせん構造を持つ Spiral Graph として表す（図 9 中央左）．以後，K 重らせん構造を持つ Spiral Graph の K 個のらせんを，便宜上，上から トラック1, トラック2, ..., トラックK と呼ぶことにする．

Spiral Graph を用いると，レジスタ割付は，この K 個のトラックの上に変数の生存期間を次々と巻き付けていくという問題になる（図 9 中央右）．すべて巻き付けたなら，スライド命令を考慮してグラフの形状を復元すればよい（図 9 最右）．

Spiral Graph におけるいくつかの用語を定義する．

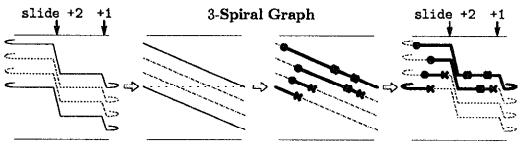


図9 Spiral Graphの考え方
Fig. 9 Concept of Spiral Graph.

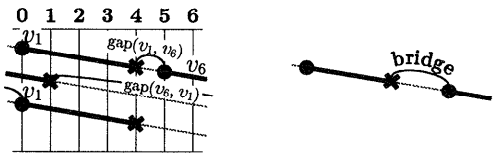


図10 変数間の隙間
Fig. 10 'gap' between variables.

定義5（始点，終点） 変数の生存期間 $v_i = [s_i, e_i)$ において，s_i $(0 \leq s_i < II)$ を変数 v_i の**始点**，e_i $(s_i < e_i)$ を**終点**という．■

定義6（開始，終了ステップ） 変数の生存期間 $v_i = [s_i, e_i)$ において，$s_i \bmod II$，$e_i \bmod II$ をそれぞれ v_i の**開始ステップ**，**終了ステップ**といい，$\mathrm{SS}(v_i)$，$\mathrm{ES}(v_i)$ と書く．■

定義7（変数間の隙間） 2変数 v_i と v_j において，$\{\mathrm{SS}(v_j) - \mathrm{ES}(v_i)\} \bmod II$ を v_i から v_j までの**隙間**といい，$\mathrm{gap}(v_i, v_j)$ と書く．隙間は非対称であることを注意しておく（**図10** 左）．■

以下，説明を簡潔に記述するため，v_i と v_j の隙間が小さければ，v_i から v_j までは**近い**といい，大きければ**遠い**ということにする．

4.3.2 Short Bridge Algorithm

簡単のため，まずは1トラックのSpiral Graph（1-Spiral Graph）から考える．先に触れたFat Cover Algorithmは，端的には，繰返しをまたぐ変数と組み合わさって周をなす変数を1つにまとめて彩色するアルゴリズムである．ところが，1-Spiral Graphは1本の連続したらせん状のグラフであるから，繰返し1回ごとに区切って考えるよりも，後に詳述するように連続して割り付けるほうが自然である．そこで，我々はFat Cover Algorithmとはまったく異なるアプローチのレジスタ割付法を考案し，Short Bridge Algorithmと名付けた．そのストラテジは非常に単純である．

Spiral Graphを導入すると，占有レジスタ数を少なくせよという問題は，生存期間の巻き付け順を適当に定めて隙間の総和を小さくし，らせんの周回数を少なくせよという問題になる．Short Bridge Algorithmの基本ストラテジは，すでに巻き付けられている変数から次に巻き付けるべき変数までの隙間が必要最小限になるように次の変数を選ぶというものである．これは，すでに巻き付けた変数の終点と，次に巻き付ける変数の始点との間をなるべく短い橋で短絡していこうとするイメージである（図10右）．

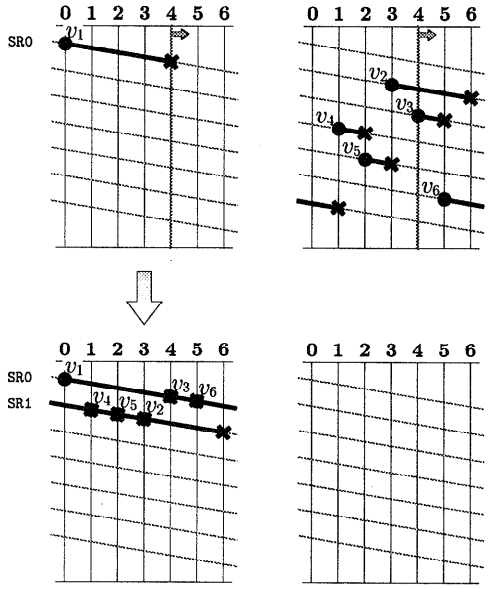


図11 1-SHORTBRIDGEによるレジスタ割付のようす
Fig. 11 Register allocation via 1-SHORTBRIDGE.

アルゴリズム4（1-SHORTBRIDGE）

(1) 変数 $v_1, \ldots, v_N$ の中から，始点の最も小さいものを選び，トラック1に巻き付ける．

(2) 残る変数の中から，トラック1上に最後に巻き付けられた変数に最も近い変数を選び，開始ステップを維持して巻き付ける．

(3) (2)を，巻き付けていない変数がなくなるまで繰り返す．■

このアルゴリズムによってレジスタを割り付けているようすを，左側を彩色済の変数，左側を未彩色の変数として**図11** に示す．

次に1-SHORTBRIDGEを K トラックに拡張するが，その際には各トラックになるべく均一に変数が割り付けられるように配慮しなければならない．特定のトラックのみが長くなりすぎると，**図12** 左のように無駄な空き領域を生じてしまうからである．

この問題を解決するには，各トラックが均等に伸長するようにすればよく，次のようなアルゴリズムで解決できる．また，より狭いレジスタウィンドウに収まるようにするためには，最後にトラックを長い順に並べ換えるのがよい（図12右）．

アルゴリズム5（K-SHORTBRIDGE）

(1) 変数 $v_1, \ldots, v_N$ の中から，始点の小さい順

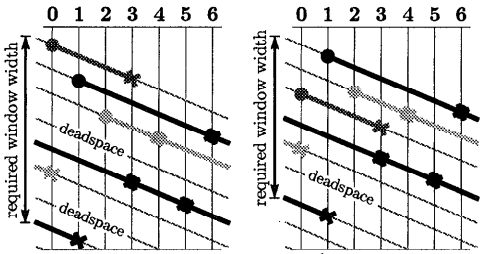


図12 K トラックの Spiral Graph におけるデッドスペース
Fig. 12 Dead spaces in K-Spiral Graph.

に K 個の変数を選び，それぞれトラック $T_k\,(1 \le k \le K)$ に巻き付ける．

(2) 残る変数の中から，各トラック上に最後に巻き付けられた変数に最も近い変数をそれぞれ候補として列挙する．候補間には重複があってもよい．

(3) すべての候補を巻き付けたと仮定する．この状態で全長が最も短いトラック上の候補を1つ確定して，残りはすべて破棄する．

(4) (2)～(3)を，巻き付けていない変数がなくなるまで繰り返す．

(5) K 本のトラックを，長い順にソートする． ■

5. preload 命令への対応

演算速度に比べて主記憶アクセスのレイテンシが大きいマシンでは，load 命令を発行してからレジスタに値が代入されるまで数十マシンサイクルを要する．これは，遅延を見積もって，先々必要になるデータを数回前の繰返しで読み込んでおくことによって隠蔽できる．これを有効に実現するための命令が preload である．

preload 命令は，単純には load 命令と同じ命令と考えてよいが，ウィンドウの外側にあるレジスタもデータ転送の対象にできるという特徴を持つ．これによってレジスタ数を圧迫せずにパイプライン処理を行い，主記憶アクセスレイテンシを隠蔽することができる．このことを，**図13** を用いて説明する．図13左は，アルゴリズム2による図4のレジスタ割付結果である．

ここで，図13右のように，ロード対象のレジスタ番号を `SR(-1)` にすると，これはループ末尾の `slide +1` 命令で `SR0` にリネームされる．この値を本来の演算に使うことにすれば，ループの繰返し1回分の時間的余裕ができたことになる．仮にレイテンシがこれより大きい場合には，`SR(-2)`,`SR(-3)`,... をロード対象とすればよい．一般に，主記憶アクセスレイテンシを L，イテレーション立ち上げ間隔を II とすると，`SR(`$-\lceil L/II \rceil$`)` にロードすればよいことが分かる．preload 命令でウィンドウの外側にロードできることによって，何周前の繰返しでロードしようとも占有レジスタ数が増えることはない．

```
do {                          do {
0:  load  SR0                 0:  preload SR(-1)
1:  add   SR1, SR0 -> SR1     1:  add   SR1, SR0 -> SR1
2:  mult  SR1, SR1 -> SR0     2:  mult  SR1, SR1 -> SR0
3:  add   SR1, SR0 -> SR1     3:  add   SR1, SR0 -> SR1
4:  mult  SR0, SR1 -> SR0     4:  mult  SR0, SR1 -> SR0
5:  add   SR1, SR0 -> SR0     5:  add   SR1, SR0 -> SR0
6:  store SR1                 6:  store SR1
    slide +1                      slide +1
}                             }
```

図13 preload 命令を用いた主記憶アクセスレイテンシの隠蔽
Fig. 13 Latency hiding with preload instruction.

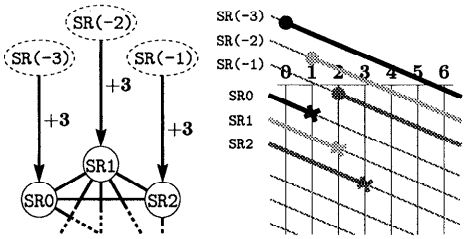


図14 preload 命令の性質を考慮したレジスタ割付
Fig. 14 Register allocation for preload instruction.

ロード対象の変数が K 個ある場合には，**図14** のように，`slide` K 命令を用いてスライド量を K にすることで解決できる．このときは，ロード対象の K 個の変数を `SR(`$(-K\lceil L/II \rceil)+0$`)` ～ `SR(`$(-K\lceil L/II \rceil)+(K-1)$`)` にロードすればよく，これによってウィンドウ内に現れたときのレジスタ番号は `SR0` ～ `SR(`$K-1$`)` になる．

preload 命令を考慮したスライドレジスタ彩色アルゴリズムは，以下のようになる．

アルゴリズム 2′ (Slide Coloring Algorithm)

(1) preload 対象の K 個の変数を，頂点の度数の大きい順に `SR0` ～ `SR(`$K-1$`)` で彩色する．

(2)～(6)は，アルゴリズム2の(1)～(5)と同様である． ■

一方，Short Bridge Algorithm は，preload 命令を考慮した変数の生存期間の定義を Spiral Graph に与えることによって，preload 対象の変数を特別視することなく統括的に扱えるアルゴリズムになる．

定義 8 (定義点，定義ステップ) 変数の生存期間 $v_i = [s_i, e_i)$ を，次の条件で3項からなる表現 $v_i = [d_i, s_i, e_i)$ に拡張する．

$$d_i \equiv \begin{cases} s_i\text{から遡った，preload 命令の発行時刻} \\ \qquad (v_i \text{ が preload 対象の変数のとき}) \\ s_i \quad (v_i \text{ が preload 対象の変数でないとき}) \end{cases}$$

d_i を変数 v_i の**定義点**とよぶ．また，$d_i \bmod II$ を

```
   do {
0:  preload  v1
1:  add      v6, v1 -> v4
2:  mult     v4, v4 -> v5
3:  add      v5, v1 -> v2
4:  mult     v1, v2 -> v3
5:  add      v2, v3 -> v6
6:  store    v2
    slide +1
   }
```

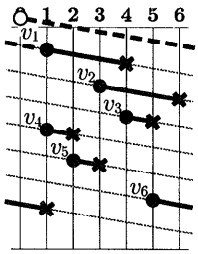


図15 preloadでロードされた変数の生存期間
Fig. 15 Live range of preloaded variables.

$\mathrm{DS}(v_i)$ と書き，v_i の**定義ステップ**とよぶ． ■

$0 \leq s_i < II$ と，$d_i \ll s_i$ より，定義点は，preload対象の変数では一般に負の値，そうでない変数については非負の値をとる．たとえば，図13右のようにpreload命令を用いて v_1 を1回前の繰返しでロードする場合，$0 \leq s_1 < II$ より，$v_1 = [-7, 1, 4)$ と表せる．v_2 は $v_2 = [3, 3, 6)$ と表せる（**図15**）．したがって，定義点の小さい変数から選び出せば自然にpreload対象のものが先に選ばれるから，定義7とアルゴリズム5中の手順(1)を変更するだけで，preload命令に対応したアルゴリズムが完成する．

定義7′（変数間の隙間） 2変数 v_i と v_j において，$\{\mathrm{DS}(v_j) - \mathrm{ES}(v_i)\} \bmod II$ を v_i から v_j までの**隙間**といい，$\mathrm{gap}(v_i, v_j)$ と書く． ■

アルゴリズム5′（Short Bridge Algorithm）

(1) 変数 $v_1, \ldots, v_N$ の中から，**定義点**の小さい順に K 個の変数を選び，それぞれトラック $T_k\,(1 \leq k \leq K)$ に巻き付ける．

(2)～(5)は，アルゴリズム5と同様である． ■

定義8，定義7′により，アルゴリズム5′は，preload対象の変数がない場合にもそのまま適用できる．

6. 実　　験

以上，スライドウィンドウおよびpreload命令を考慮したレジスタ割付法としてSlide Coloring AlgorithmとShort Bridge Algorithmを提案したが，その2つのアルゴリズムの比較実験を行った．

6.1 Livermore Fortan Kernelによる評価

評価実験のために，実験用Fortranコンパイラおよびレジスタ割付器を用意した．実験用コンパイラは，与えられたソースプログラムに含まれるループをソフトウェア・パイプライニングによって疑似ベクトル化し，スケジューリングの結果得られた変数の生存期間を $v_i = [d_i, s_i, e_i)$ の形式でファイルにダンプして終了する．また，割付器は，Slide Coloring AlgorithmおよびShort Bridge Algorithmを実装したものであり，この中間出力を入力としてレジスタ割付を行うものである．

Livermore Fortran Kernel（LFK）のソースは，24個のカーネルおよびそれらで用いるサブルーチンからなる．このソースを実験用コンパイラにかけたところ，slide命令を用いてスケジューリングされたループが55個得られた．得られた55個の例題についてSlide Coloring AlgorithmおよびShort Bridge Algorithmによってレジスタ割付を行い，総当たり法によって得た最適解との比較を行った．使用するレジスタはある与えられたスライドウィンドウの幅に収まっている必要があるから，比較は単純に占有レジスタ数で行うわけにはいかない．たとえばあるサンプルに対する占有レジスタ数が20であるとする．このとき，占有レジスタ間に図12のような空きレジスタがある場合は，幅20のウィンドウに収めることはできない．したがって，アルゴリズムの評価をするためには，占有レジスタ間の空きレジスタも占有したものとして，占有ウィンドウ幅の比較にする必要がある．

実験結果を**表1**に示す．表中に W_{max} を併記している理由は後に述べる．また，（CP-PACS）は，CP-PACSで現在稼働中のFortranコンパイラによるレジスタ割付結果を示したものである．CP-PACSのコンパイラと実験用コンパイラには細かな差異があるため一概に数値を比較することはできないが，CP-PACSのコンパイラもSlide Coloring Algorithmを採用しているため，参考までに併記してある．

2つのアルゴリズムを比較すると，Slide Coloring Algorithmは最適解を与えたものの割合は小さいが，最適解からの差分は小さく抑えられている．Short Bridge Algorithmはその逆に，最適解を与えたものの割合が大きいが，最適解からの差分が大きくなった例もみられる．LFKでの実験は総当たり法で最適解を求めうる程度のサンプル数であり，両者の優位性を比較するには量的に乏しいが，2つのアルゴリズムがともに90%以上の精度で最適解を与えていることが読み取れる．

6.2 ループプログラム自動生成ツールによる評価

より多くのサンプルで傾向を調べるため，我々の研究室で作成したループプログラム自動生成ツールによって10,000例のループを生成した．得られたサンプルは**図16**のような加算または乗算を含むものであり，分布は**図17**のようであった．

図17の棒グラフは，10,000個のサンプルを変数の数で分類したものである．たとえば，変数を60個含むようなサンプルは10,000個中9つ生成されている．グラフ上の点 ∘ は，変数の数と W_{max}（ループ中での

表1 レジスタ割付結果 (LFK)
Table 1 Experimental Result (LFK).

最適解との差分	−2	−1	±0	+1	+2	+3
W_{max}	2 (3.64%)	8 (14.55%)	45 (81.81%)			
Slide Coloring			51 (92.73%)	3 (5.45%)	1 (1.82%)	
Short Bridge			53 (96.36%)		1 (1.82%)	1 (1.82%)
(CP-PACS)			46 (83.63%)	8 (14.55%)	1 (1.82%)	

```
tmp1_1 = tmp1_0;
tmp101_1 = tmp101_0;
tmp201_1 = tmp201_0;
tmp301_1 = tmp301_0;
tmp401_1 = tmp401_0;
tmp1_0 = (array2[i])+((array3[i])*(tmp1_1));
tmp101_0 = (array102[i])+((array103[i])*(tmp101_1));
tmp201_0 = (array202[i])+((array203[i])*(tmp201_1));
tmp301_0 = (array302[i])+((array303[i])*(tmp301_1));
tmp401_0 = (array402[i])+((array403[i])*(tmp401_1));
array0[i] = tmp1_0;
array100[i] = tmp101_0;
array200[i] = tmp201_0;
array300[i] = tmp301_0;
array400[i] = tmp401_0;
```

図16 生成されたループプログラム (本体) の例
Fig. 16 An example of generated loop program (body part).

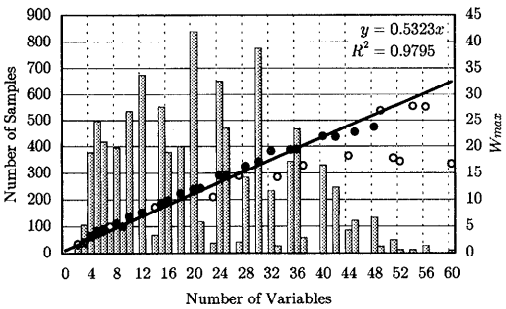


図17 変数の数と W_{max} の関係
Fig. 17 Relation between number of variables and W_{max}.

同時生存変数の最大値) との相関を示している. たとえば, 点 (60, 16.7) は, 変数を 60 個含むサンプルにおいては, W_{max} の平均 $\overline{W_{max}}$ が 16.7 であったことを示している. 棒グラフが 100 (全サンプル数 10,000 の 1%) を超えているものについては黒で塗りつぶし, 1 次回帰を行った. 回帰直線の式 $y = 0.5323x$ ($\cong x/2$) は, W_{max} がおおよそ全変数の半分の値であることを示している.

これらのサンプルを実験用の簡易 C コンパイラでコンパイルした. このコンパイラも与えられたソースプログラムに含まれるループをソフトウェア・パイプライニングによって疑似ベクトル化し, スケジューリングの結果得られた変数の生存期間をファイルにダンプして終了する. こうして得られた中間出力に対し, 2 つの割付器をそれぞれ適用した. 近似アルゴリズムの評価

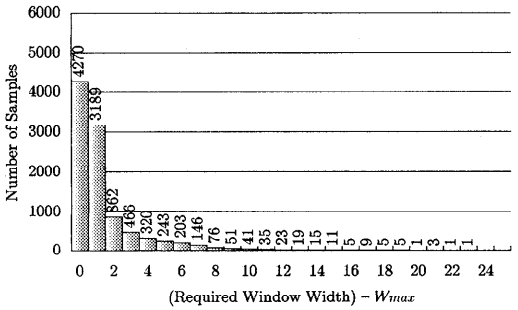


図18 Slide Coloring Algorithm
Fig. 18 Experimental result (via Slide Coloring Algorithm).

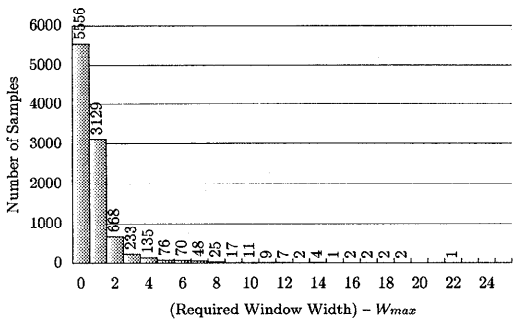


図19 Short Bridge Algorithm
Fig. 19 Experimental result (via Short Bridge Algorithm).

には, 本来であれば最適解との差分を用いるのが最良であるが, 10,000 例のサンプルのすべてに対して最適解を総当たり法で求めるのは現実的でない. そこで, 個々のサンプルの W_{max} 値を占有ウィンドウ幅の下界として用い, そこからの差分で評価を行った. このとき, W_{max}, 占有ウィンドウ幅の最小値 (以下, 最適解とよぶ), 提案したアルゴリズムによる近似解との間には $W_{max} \leq$ 最適解 $\leq$ 近似解 の関係があるから, もし $W_{max} =$ 近似解 であれば, はさみうちの定理により 3 つの値が一致し, 近似解の最適性を保証できる. 一方, $W_{max} \neq$ 近似解 であるときは, 近似アルゴリズムが冗長な割付をしているか, W_{max} が下界として過小評価であるかのいずれかである.

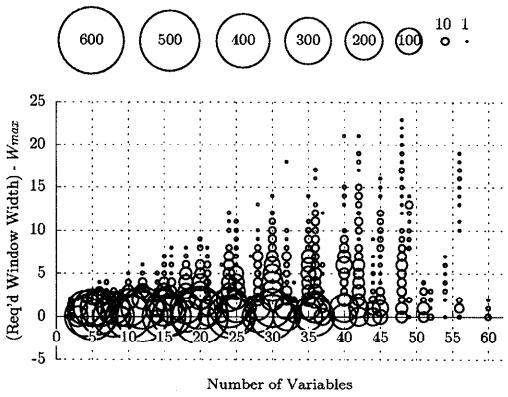


図20 Slide Coloring Algorithm
Fig. 20 Experimental result (via Slide Coloring Algorithm).

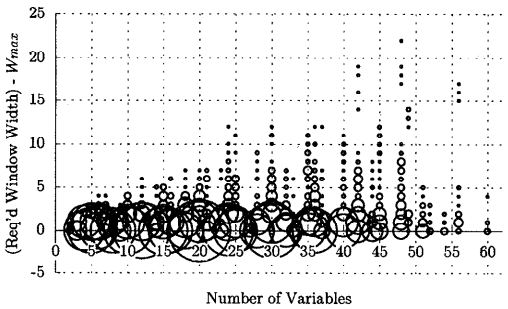


図21 Short Bridge Algorithm
Fig. 21 Experimental result (via Short Bridge Algorithm).

結果を**図18**および**図19**に示す．Slide Coloring Algorithmでは少なくとも42%，Short Bridge Algorithmでは55%が最適解であるといえ，Short Bridge Algorithmの方が効果的に割付できていることが分かる．最適解の割合がLFKでの値に比べて小さいことの原因としては，表1から分かるように，W_{max}は最適解よりも小さい値をとることがあるため，下界として過小評価であることが考えられる．

参考までに，全サンプルにおける傾向を一覧できるようにしたものが，**図20**および**図21**である．これらのグラフは，x座標に変数の数，y座標に占有ウィンドウ幅からW_{max}を減じた値をとり，サンプル数を円の面積で示している．上述したようにW_{max}だけでは最適性の判定を行うことができないが，W_{max}からの差分が小さければ最適解からの差分も小さいから，グラフの上部に多くの点が現れないほど効果的な割付であるといえる．

Short Bridge Algorithmの方がやや効果的な割付ができている理由としては，変数どうしが干渉する/しないの2値のみを保存するレジスタ干渉グラフよりも，変数どうしの隙間を保存するSpiral Graphの方が情報量が多いこと，また，Short Bridge AlgorithmがSpiral Graphの持つ隙間の情報を活用するように構成されていることが考えられる．

7. まとめ

本論文では，スライドウィンドウを考慮したレジスタ割付法として，(A)スライドレジスタ干渉グラフおよびSlide Coloring Algorithmからなるフレームワーク，(B) Spiral GraphおよびShort Bridge Algorithmからなるフレームワークを提案した．LFKによる実験では両者ともに9割以上のサンプルに対して最適解を与えた．また，ループプログラム自動生成ツールによる実験では，少なくとも(A)が42%，(B)が55%のサンプルに対して最適解を与えていることが分かった．ループプログラム自動生成ツールにおける最適解の割合がLFKに比べて小さい原因としては，評価に用いた下界W_{max}が過小評価であることが考えられる．このことについては別途報告する予定である．

Spiral GraphとShort Bridge Algorithmからなるフレームワークは，隙間の情報を持ち，その情報量を活かせるアルゴリズムであること，および，ループ1周の合計スライド量のみに着目するだけでよいこと，preload変数をも自然に扱えるアルゴリズムであることの柔軟性などから，将来有望であると考えられる．このフレームワークは，現在，CP-PACSのコンパイラへの組み入れが検討されている段階である．

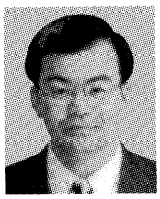
現在は，度数の同じ頂点の彩色順や，同じ長さのトラックの確定順などまでは特に定めていない．これらの点も含め，今後は，精度を高めるための手法や，その評価に用いることのできる的確な下界を見つけるために，スライドレジスタの性質そのものについてさらに追求していく予定である．また，その一方で，レジスタスピルや条件分岐，グローバルレジスタとの統合戦略など，現実的な要求をみたすアルゴリズムにしていくことも考えたい．

謝辞 超並列計算機CP-PACSの開発，製作に協力いただいている（株）日立製作所に謝意を表したい．CP-PACSプロジェクトは文部省科学研究費補助金（創成的基礎研究）07NP0401による．

参考文献

1) 中澤喜三郎，中村　宏，朴　泰佑：超並列計算機CP-PACSのアーキテクチャ，情報処理，Vol.37, No.1, pp.18–28 (1996).
2) 位守弘充，中村　宏，朴　泰佑，中澤喜三郎：スライドウィンドウ方式による疑似ベクトルプロセッサ，情報処理学会誌，Vol.34, No.12, pp.2612–2623 (1993).
3) 中田育男，山下義行，小柳義夫：超並列計算機CP-PACSのソフトウェア，情報処理，Vol.37, No.1, pp.29–37 (1996).
4) Lam, M.: Software Pipelining: An Effective Scheduling Technique for VLIW Machines, *Proc. SIGPLAN '88 Conf. on PLDI*, pp.318–328 (1988).
5) Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E. and Markstein, P.W.: Register Allocation via Coloring, *Computer Lang.*, Vol.6, pp.47–57 (1981).
6) Chaitin, G.J.: Register Allocation and Spilling via Graph Coloring, *Proc. SIGPLAN '82 Symp. on Compiler Construction, ACM SIGPLAN Notices*, pp.98–105 (1982).
7) Chow, F.C. and Hennessy, J.L.: Register Allocation by Priority-Based Coloring, *Proc. SIGPLAN '84 Symp. on Compiler Construction*, pp.222–232 (1984).
8) Hendren, L.J., Gao, G.R., Altman, E.R. and Mukerji, C.: *A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs*, LNCS, No.641, pp.176–191, Springer-Verlag (1992).
9) Hewlett Packard: PA-RISC 1.1 Architecture and Instruction Set Reference Manual (1990).
10) 日立製作所：ハードウェア・オペレーティング・マニュアル，SR2201並列プロセッサ.

(平成9年11月19日受付)
(平成10年7月3日採録)



秡川　友宏（学生会員）

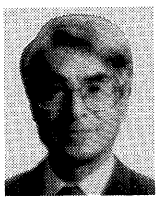
1970年生．1995年筑波大学第1学群自然学類卒業．同年同大学大学院工学研究科入学．

添野　元秀（学生会員）

1971年生．1994年筑波大学第3学群情報学類卒業．同年同大学大学院理工学研究科入学．1996年同大学大学院工学研究科編入．

山下　義行（正会員）

1959年生．1982年大阪大学理学部物理学科卒業，日立マイクロコンピュータ・エンジニアリング（株）入社．1986年退社．1987年筑波大学大学院博士課程工学研究科電子・情報工学専攻入学．1989年退学，東京大学大型計算機センター助手．1992年筑波大学電子・情報工学系講師．1995年～同助教授．工学博士．プログラミング言語，コンピュータ・グラフィクスの研究に従事．日本ソフトウェア科学会会員．



中田　育男（正会員）

1935年生．1958年東京大学理学部数学科卒業．1960年同大学大学院修士課程修了．1960～1979年（株）日立製作所中央研究所システム開発研究所勤務．1979～1997年筑波大学電子・情報工学系教授．1997年～図書館情報大学図書館情報学部教授．理学博士．プログラム言語，言語処理系，ソフトウェア工学などに興味を持っている．著書「コンパイラ」（産業図書），「基礎FORTRAN」（岩波書店），「コンパイラ」（オーム社）．日本ソフトウェア科学会，電子情報通信学会，ACM，IEEE各会員．