VLIW マシンのための非数値計算応用向き広域命令スケジューリング手法

安藤 秀樹[†1] 中西 知嘉子[†2]
原 哲也[†3] 中屋 雅夫[†4]

VLIW マシンにおいて非数値計算応用プログラムに対して高い性能を達成するには，任意のパスからの命令移動を行う広域命令スケジューリングが必要である．しかしこれまでの手法は，スケジューリングのウィンドウの出口近くにおける並列度の低下や，繰返し確率の高くないループに対する最適化に対して，十分な考慮がなされておらず，最適化が十分ではなかった．本論文では，これらの問題に対する最適化手法を提案する．広域命令スケジューリングでは，分岐を越える投機的命令移動が重要であるが，我々は，これを支援するプレディケーティングと呼ぶハードウェア機構を提案している．プレディケーティングは投機的命令移動に関する制限を取り除くので，コンパイラは最適化能力を最大限発揮できる．プレディケーティングを備えた4命令発行の VLIW マシンに対する評価の結果，本スケジューリング手法を用いれば，17%性能を改善することができ，その結果，スカラ・マシンの 2.40 倍の性能が達成できることを確認した．

Global Instruction Scheduling of Non-numerical Applications for a VLIW Machine

HIDEKI ANDO,[†1] CHIKAKO NAKANISHI,[†2] TETSUYA HARA[†3]
and MASAO NAKAYA[†4]

Global instruction scheduling capable of code motion along any control path is required for achieving good performance in a VLIW machine. Previous scheduling techniques, however, have problems of parallelism decrease near window exits and poor optimization for a loop which does not iterate many times. This paper proposes instruction scheduling techniques for these problems. Speculative code motion is strongly required for global instruction scheduling. Recently we proposed a hardware support for speculative execution which we call predicating. Since predicating removes constraints imposed on speculative code motion through the compiler, the compiler is allowed to best use its optimizing ability. Our evaluation results show that our optimizations improve performance in a 4-issue VLIW machine with predicating by 17%. Consequently, we confirmed 2.40x speedup over a scalar machine.

1. はじめに

VLIW マシンの性能を向上させるには，命令の並べ変えによりコードの中に存在する並列性を最大限抽出することが重要である．命令間の依存と与えられたハードウェア資源の制約を満足し，命令の実行順序を定めることを**命令スケジューリング**という．一般に，非数値計算応用プログラムでは，基本ブロック内の命令レベルの並列度は非常に小さいので[1),2)]，基本ブロック境界を越えてスケジューリングを行うことが必要である．基本ブロックを越えてスケジューリングを行うことを，**広域命令スケジューリング**と呼び，このとき行われる命令移動を**広域命令移動**と呼ぶ．

これまで，多くの広域命令スケジューリング手法が提案されてきた．初期の手法は，隣り合う基本ブロック間での最適化を繰り返すことによりプログラム全体の最適化を実現する手法であった[3),4)]．しかし，近年提案されている広域命令スケジューリング手法の多くは，複数の基本ブロック境界を越える命令移動を一度

†1 名古屋大学大学院工学研究科電子情報専攻
Department of Information Electronics, School of Engineering, Nagoya University
†2 三菱電機株式会社システム LSI 開発研究所
System LSI Laboratory, Mitsubishi Electric Corporation
†3 三菱電機株式会社マイコン・ASIC 事業統括部
Microcomputer & ASIC Division, Mitsubishi Electric Corporation
†4 三菱電機株式会社システム LSI 開発部
System LSI Development Division, Mitsubishi Electric Corporation

で行う手法を採っている[5)〜8)]．これを行うために，スケジューリング前にスケジューリング対象とする複数の基本ブロックを選択し，その中の命令の依存関係を計算する．初期のスケジューリング手法では，ブロック間の命令移動が必ずしもプログラム全体の性能改善につながらないという欠点があるが，近年のこれらのスケジューリング手法では，複数の基本ブロックにまたがる依存情報を得ているので，性能を改善するために移動すべき命令を正確に発見できる．一般に，スケジューリング対象であるコードの一部分のことを，**命令ウィンドウ**，あるいは，単に**ウィンドウ**と呼ぶ．

このようにウィンドウを選択しスケジューリングを行う手法は優れているが，問題が2つある．1つは，ウィンドウの出口に近づくほどスケジューリング対象の命令の数は減少するので，ウィンドウの出口近くほど並列度が低下する問題である．この問題に対しては従来，コードの複写を行いウィンドウを拡大することによって緩和していた（たとえば，ループ・アンローリングや**分岐先拡張**[9)]）．これらの方法は有効ではあるが，コード量やコンパイル時間が増加する問題があり，満足できる方法ではない．

もう1つの問題は，繰返し確率が高くないループに対する最適化である．一般に，ループに対しては，ループ・アンローリングやソフトウェア・パイプライニングなどのループ最適化技術が知られている．これらの技術は，ループの繰返し確率が高ければ有効である．しかし，そうでなければ，有効性が減少しコード量の増加がオーバヘッドとなる．数値計算応用では，大きな配列に対する計算など繰返し確率の高いループが多い．しかし，非数値計算応用では，短いリストへの操作など繰返し確率の高くないループも多い．したがって，そのようなループに対する最適化手法が必要である．

本論文では，上記2つの問題を解決する**リージョン最適化スケジューリング**と呼ぶ命令スケジューリング手法を提案し，その有効性を評価する．以下，2章では，本スケジューリング手法を適用するアーキテクチャについて説明する．3章では，これまで提案されている広域命令スケジューリング手法についてまとめる．4章では，リージョン最適化スケジューリング手法を提案する．5章で評価結果を述べ，最後に，6章で結論を述べる．

2. アーキテクチャ上の仮定

本スケジューリング手法が仮定しているアーキテクチャは，**プレディケーティング**と呼ぶ投機的実行を支援する機構を持つVLIWマシン・アーキテクチャである．本章では，最初に投機的実行における問題について説明し，次に，この問題を解決するプレディケーティングについて簡単に説明する．なお，詳細については，本論文の範囲を越えているので，文献10)，11)を参照してほしい．

2.1 投機的実行における問題

投機的実行には，一般に，次の2点に関して問題がある．

- プログラムの意味の維持
- 投機的命令の起こした例外の処理

分岐を越えて命令移動を行い，その命令の書き込みが他の分岐方向で使用される値を破壊する場合，プログラムの意味が維持できない．命令移動によってプログラムの意味が壊される場合，この命令移動は**不正**（**illegal**）であるという．コンパイラあるいは何らかのハードウェア機構は，不正な命令移動に対して，プログラムの意味を維持するように対処しなければならない．

投機的に実行された命令が実行中に例外を起こした場合，その例外を，通常の例外と同様に即座に処理を行うと，実行を不正に停止させるか，あるいは，性能を大きく低下させる．これは，その例外処理が真に必要かどうか不明であるにもかかわらず行われるためである．投機的命令が例外を起こす可能性がある場合，その命令は**危険**（**unsafe**）であるという．また，投機的な命令によって引き起こされる例外のことを**投機的例外**と呼ぶ．投機的例外が生じた場合は，例外を起こした命令の実行が真に必要であることが分かるまで，例外の処理を延期する必要がある．例外の処理が延期されるので，例外を起こした命令の誤った結果を用いた命令の実行結果は正しくない．したがって，例外から実行の再開が必要な場合，例外を起こした命令の再実行だけでなく，その命令の結果を使用する命令を再実行し，マシン状態（命令の実行結果により構成される状態）を正しくしなければならない．

2.2 プレディケーティング

プレディケーティングでは，すべての命令はプレディケートを持ち，次のような形式である．

プレディケート ? 操作

プレディケートは，その命令が依存する分岐条件の論理式である．命令の意味は，「プレディケートが真であるときのみ，操作部で示された操作の結果が有効となる」である．

プレディケートが参照する分岐条件は，汎用レジスタではなく，CCR（Condition Code Register）と

```
      if (c1)
i1:      r1 = load r2;
         if (c2)
i2:        r3 = r1 + 1;
      } else {
i3:      r4 = r1 & r2;
      }
```

(a)

```
i1': c1     ? r1 = load r2
i2': c1&c2 ? r3 = r1 + 1
i3': !c1    ? r4 = r1 & r2
```

(b)

図1 プレディケーティングにおけるコード例
Fig. 1 Code example with predicating.

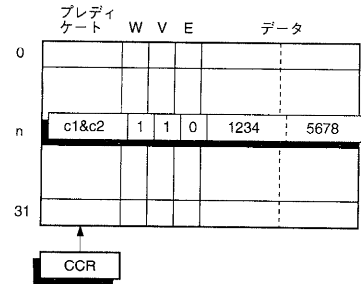


図2 プレディケート付きレジスタ・ファイル
Fig. 2 Predicated register file.

呼ぶレジスタに記憶する．CCR は複数のエントリを持ち，各エントリは異なる分岐条件の値を記憶する．CCR のエントリ数と等しいかそれ以下の数の分岐を越える投機的移動が可能である．

図1にプレディケーティングにおけるコードの例を示す．図1(a) が通常のマシンに対するコードで，図1(b) がプレディケーティングにおける対応するコードである☆．命令 **i1** と **i1'**，**i2** と **i2'**，**i3** と **i3'** が対応している．*cn* はCCR の第 *n* エントリを示す．命令にプレディケートを付加することにより，複数の制御パスは単一の制御パスにされる．

マシンは投機的実行を行うために，**逐次的状態**と**投機的状態**の2つのマシン状態を持つ．逐次的状態は，制御依存が解消している命令の実行結果により構成されるマシン状態である．投機的状態は，制御依存が解消していない命令の実行結果により構成されるマシン状態である．

命令の発行時，プレディケートが評価される．その結果，値が真であれば，通常のマシンの命令と同様に，実行を行い逐次的状態を更新する．これを**逐次的実行**と呼ぶ．もしも，プレディケートの値が偽であれば，発行時点で命令は無効化される．以上の動作は，従来のプレディケート実行[7),12),13)]と同じである．プレディケーティングが従来のプレディケート実行と異なる点は，命令の発行時点において，プレディケートが参照する分岐条件の値がまだ定義されておらず，その結果，プレディケートの値の真偽が不明な場合も実行（投機的実行）を行い終了する点である．投機的実行の結果は，投機的状態に書き込む．

レジスタ値に関する投機的状態を保持し，適切に逐次的状態を更新するために，レジスタ・ファイルは特別な構造を持っている．**図2**にレジスタ・ファイルの構成を示す．各エントリは，2つのデータ記憶部，1つのプレディケート記憶部，3つのフラグ（W，V，E）を持つ．2つのデータ記憶部のどちらか一方は，当該エントリに対応するレジスタ値に関する逐次的状態を保持し，他方は投機的状態を保持する．逐次的状態を保持しているレジスタを**逐次的レジスタ**と呼び，投機的状態を保持しているレジスタを**投機的レジスタ**と呼ぶ．フラグ W は，どちらのデータ記憶部が投機的レジスタであるかを示す．フラグ V は，フラグ W で指定された投機的レジスタの保持する値が有効であることを示す．フラグ E は，そのエントリに書き込みを行った命令が実行中に例外を起こしており，かつ，その処理が延期されていることを示す．

投機的実行の結果は，投機的レジスタに書き込まれる．この際，フラグ V がセットされ，その命令のプレディケートはプレディケート記憶部に書き込まれる．各エントリのプレディケート記憶部は，CCR を参照しプレディケートの値を毎サイクル評価する専用のハードウェアを持つ．プレディケートが真と評価された場合は，投機的レジスタに格納された値は逐次的レジスタに移動され，偽と評価された場合は，投機的レジスタに格納された値は無効化される．投機的状態にある実行結果で逐次的状態を更新することを，投機的実行結果の**コミット**という．プレディケートが真と評価された場合，フラグ W を反転させ，フラグ V をリセットする．これは投機的実行結果のコミットに相当する．また，プレディケートが偽と評価された場合，フラグ V をリセットする．これは投機的実行結果の無効化に相当する．

投機的例外が発生した場合は，実行結果の書き込みにおいて，フラグ V をセットしプレディケートを書き込むほか，フラグ E をセットする．後に，フラグ V とフラグ E の両方がセットされているエントリのプレディケートが真と評価された場合，レジスタ・ファイルは投機的例外を検出したことを知らせる信号を出す．この後，投機的状態をすべて破棄し，現在のウィ

☆ このコードはスケジューリングによって並べ変えられるので，図1(a) のコードが一意に図1(b) のコードになるわけではない．

ンドウ☆の先頭より投機的例外の検出点まで，まだコミットされていない値を生成した命令を選択し再実行する．先に例外を起こした命令は，再実行時に再び例外を発生し，これを処理する．再実行により正しいマシン状態が再構築される．

メモリ値に対する投機的状態は，データ・キャッシュの前に置かれるストア・バッファに記憶する．ストア・バッファはFIFOで構成し，ストア命令の実行が，逐次的か投機的にかかわらず，ストア値はストア・バッファにいったん書き込まれる．FIFOの先頭の値が逐次的状態の値であれば，その値はデータ・キャッシュに書き込まれる．各エントリは，1つのデータ記憶部のほかに，レジスタ・ファイルと同様にプレディケート記憶部と3つのフラグ（W，V，E）を持つ．ただし，フラグWは，そのエントリのデータ記憶部が保持する値が，投機的状態の値であることを示す．レジスタ・ファイルの場合と同様に，フラグの操作によって投機的実行結果のコミットと無効化を行う．

3. 広域命令スケジューリング手法の背景

本章では，次章で説明するリージョン最適化スケジューリング手法の理解を助けるために，これまで提案されている広域命令スケジューリング手法についてまとめる．

初期の広域命令スケジューリング手法[3]は，隣り合うブロック間での最適化を繰返し行うことによりプログラム全体の最適化を行う手法をとっていた．これを**繰返し型**命令スケジューリング手法と呼ぶ．**パーコレーション・スケジューリング**[4]は，その代表例である．ブロック間の最適化を繰り返すことによって結果的にプログラム全体に対して命令移動が行われることとなる．この手法は，隣り合うブロックをウィンドウとする「狭い視野」の広域命令スケジューリング手法ということができる．この手法では，ブロック間での性能改善がプログラム全体の性能改善につながるとは必ずしもいえないので，最適なスケジュール結果を得ることはできない[5),8]．

これに対して，近年の広域命令スケジューリング手法は，ウィンドウを大きくとり，より「広い視野」でスケジューリングを行う．このスケジューリング手法で初期のものは，高い頻度で実行される単一の制御パス（**トレース**）をウィンドウとしスケジューリングを行うものである（たとえば，**トレース・スケジューリング**[5]や**スーパブロック・スケジューリング**[6]）．これを**トレース型**命令スケジューリング手法と呼ぶ．この手法は，繰返し型手法に比べて，広域な依存情報を持っているので，広い範囲の中からスケジューリングすべき最適な命令を発見することができる点で優れている．しかし，依然として，単一の制御パスにウィンドウが制限されているので，スケジューリングすべき最適な命令の発見において制限されている．この制限は，分岐方向の偏りが著しい数値計算応用プログラムに対しては影響は少ないが，そうではない非数値計算応用プログラムに対しては，トレース内を制御が移行する確率が低下するので，影響が大きい．

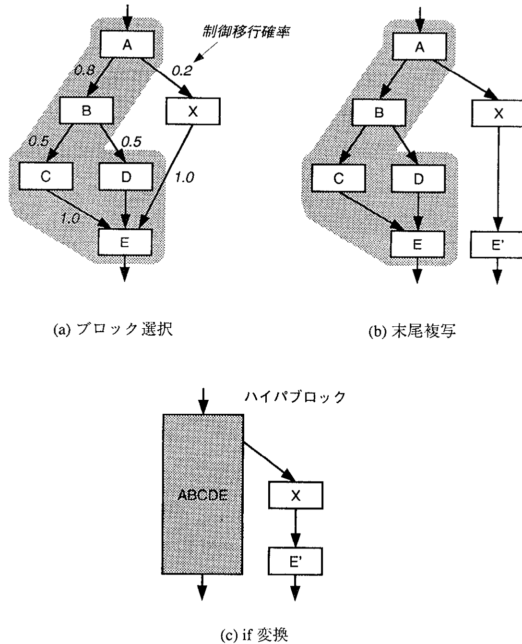
これに対して，ウィンドウ内に任意のパスを含める手法がある[7),8),14]．これを**DAG**（Directed Acyclic Graph）**型**命令スケジューリング手法と呼ぶ．DAG型では，任意のパスからの命令移動を行うので，分岐方向の偏りが少ないプログラムに対して有効である．

最初のDAG型スケジューリング手法は，Ebciogluらの広域スケジューリング**PSr**（Percolation Scheduling with Resources）[14]である．PSrでは，最初に，バック・エッジ（ループ・テールよりヘッドへのエッジ）[15]を除いたループをウィンドウとし，各基本ブロックのすべての後続ブロックからスケジュール可能な命令の集合（**unifiable-ops**）を計算する．その後，スケジュール可能な命令の中から最も高い優先度の命令を移動し，スケジューリングする．優先度は移動により越える分岐の数で定義する．移動の際，追加的にunifiable-opsを再計算することによって，広い範囲での移動可能命令の計算のコストを下げた．しかし，広域な依存情報を持っていないために，性能を向上させるための命令移動の指標に欠け，最適化が難しい．

これに対して，小松らのスケジューリング[8]は，命令の広域な依存情報をスケジューリングに利用する点で優れている．プレディケート実行を行うマシンを仮定している．小松らの手法では，ウィンドウ内の命令にプレディケートを付加し，制御依存とデータ依存を含むスケジューリング制約を表すDAGを計算する．そして，実行頻度が高く依存パス長が長い順にパスを短縮する変換を行う．ウィンドウ内のDAGを持つことによって，重要なパスを優先してスケジューリングを行うことができる．

ハイパブロック・スケジューリング[7]も，ウィンドウ内の命令のスケジューリング制約を表すDAGを作成しスケジューリングに利用する手法の1つである．ハイパブロックと呼ばれるウィンドウの形成手順を，**図3**を用いて説明する．まず，ループ・ヘッドより

☆ 制御がウィンドウを移動する度に命令アドレスがハードウェアにより記憶される．



(a) ブロック選択　(b) 末尾複写

(c) if 変換

図 3　ハイパブロック形成
Fig. 3　Hyperblock formation.

ループ内のブロックに対して，プロファイルを利用して実行頻度が高いブロックを選択し，ウィンドウとする（図 3 (a)）．次に，設定したウィンドウの先頭以外にウィンドウ外から入る制御エッジ（**サイド・エッジ**）がある場合（図 3 では X から E へのエッジ），そのエッジが入るブロック以降をすべて複写し，サイド・エッジを取り除く（図 3 (b)．E を複写）．これを**末尾複写**（**tail duplication**）[6]と呼ぶ．次に **if 変換**[16]によって命令にプレディケートを付加し，ウィンドウ内ブロックを 1 つの大きなブロックにまとめ，スケジューリングを行う（図 3 (c)）．if 変換後のブロックをハイパブロックと呼ぶ．ハイパブロック・スケジューリングでは，ループ内より実行頻度が比較的高いブロックのみをあらかじめ選択しウィンドウとすることにより，実行頻度の低い命令によって命令バンド幅や機能ユニットなどの資源が無駄に消費されることを防いでいる．また，サイド・エッジが取り除かれているので，命令移動によりウィンドウ外への補償コードの挿入が必要ないなど，最適化の適用が容易であるという特長がある．

4. リージョン最適化スケジューリング手法

我々の目的は，非数値計算応用プログラムにおいて有効なスケジューリング手法を見い出すことである．この目的のためには，DAG 型スケジューリング手法が適している．DAG 型スケジューリング手法の中でも，複数のパスにわたる複数の基本ブロックを，1 つの基本ブロックのように扱うハイパブロック・スケジューリングのアプローチは，広いウィンドウ内での最適化を単純に実現する方法として非常に有力であると考え，基本的にこのアプローチをとることとした．すなわち，

- スケジューリング前にスケジューリングの対象とするウィンドウを選択する（このウィンドウを我々はリージョンと呼ぶ）．
- サイド・エッジがある場合，末尾複写を行う．
- 命令にプレディケートを付加することによりリージョンを 1 つのブロックとし，スケジューリングを行う．

以下，提案のリージョン最適化スケジューリング手法について詳しく議論する．

4.1 概　要

リージョン最適化スケジューリング手法は，ハイパブロック・スケジューリングのフレーム・ワークの中で，さらに次の特長を持つ．

(1) ウィンドウの出口近くほど並列度が低下する問題に対処するために，リージョンの次のブロックをウィンドウに加える．このブロックを**拡張ブロック**と呼ぶ．拡張ブロック内の命令は，リージョン内の命令だけでは埋めることができなかったスケジューリング・スロットを埋めるために，限定的にリージョン内に移動する．拡張ブロックとしては，未スケジュールのブロックだけでなく，既スケジュールのブロックも選択し，既スケジュールのブロックを含めた最適化を行う．

(2) 繰返し確率の高いループとそうでないループでスケジューリング手法を切り替える．繰返し確率の高いループに対しては，ソフトウェア・パイプライニングにより最適化を行う．一方，繰返し確率の高くないループに対しては，ループのバック・エッジに沿った命令移動と，ループの外からの命令移動による利益を同時に考慮する新たな手法を導入し，最適化する．

(3) 本フレームワークとプレディケーティングに適した最適化を行う．たとえば，ウィンドウ選択により新たにループ不変[15]となった命令のループ外への投機的移動を行う．

図 4 に命令スケジューリングの概要を示す．命令スケジューリングは，基本的にループごとに，内側から外側に向かって行う☆．まず初めに，スケジューリング対象のプロシージャに対して，CFG（Control Flow

☆ 最も外側はプロシージャ・ボディである．

```
foreach PROCEDURE {
   CFG を作成；
   データフロー情報を計算；
   foreach LOOP（内側から外側に向かって） {
      while（LOOP に未スケジュールの基本ブロックがある）{
          REGION を形成；
          REGION をスケジュール；
          拡張ブロックに対するブックキーピング；
          REGION を簡約し CFG を更新；
      }
   }
}
```

図 4 リージョン最適化スケジューリングの概要
Fig. 4 Overview of region optimizing scheduling.

Graph)[15),☆]を作成し，live 情報やエイリアスなどのデータフロー情報を計算する[15)]．その後，スケジューリングされていない最も内側のループ（**現在のループ**と呼ぶ）を選択し，その中でスケジュールされていない基本ブロックを選択する．これをリージョンの入口のブロックとする．このブロックのことを，リージョンの**ヘッダ**と呼ぶ．そして，プロファイリングによる分岐予測情報を利用して，ヘッダより実行確率の高いブロックを加えることにより，リージョンを選択する．リージョンのヘッダは現在のループの中に存在するが，リージョンとしては（それが利益があるならば），ループの外のブロックも含める．その後，サイド・エッジがあれば末尾複写を行い，リージョンを形成する．

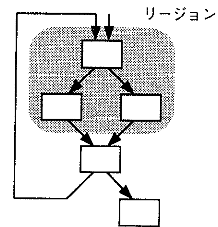
リージョンのスケジューリングにおいては，リージョンと現在のループの関係で，次のように手法を切り替える（図 **5** 参照）．

(1) リージョン選択の何らかの条件によって，リージョンがループをなさなかった場合（図 5 (a)）は，4.2〜4.5 節で説明する通常のリージョンに対するスケジューリング手法を適用する．

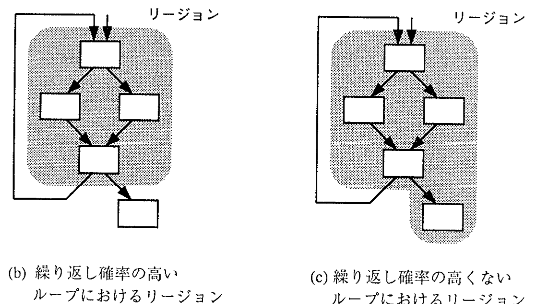
(2) リージョンがループをなす場合で，ループの繰返し確率が高いときは，ループ外のブロックの実行確率は低いため，リージョンはループ内ブロックだけで形成される（図 5 (b)）．この場合，ループ最適化が有効なので，ソフトウェア・パイプライニングを適用する．

(3) リージョンがループをなす場合で，ループの繰返し確率が高くない場合は，ループ外のブロックの実行確率が低くないので，リージョンはループの外にも広がる（図 5 (c)）．この場合，(1) の場合に行う通常のスケジューリングに加えて，4.6 節で述べる繰返し確率の高くないループに対する最適化を行う．

☆ ノードで基本ブロックを，有向エッジで制御フローを表すグラフ．



(a) ループをなしていないリージョン



(b) 繰り返し確率の高いループにおけるリージョン　(c) 繰り返し確率の高くないループにおけるリージョン

図 5 リージョンと現在のループ関係
Fig. 5 Relationship between a region and a loop.

ここで，リージョンがループをなしているとは，リージョンのヘッダが現在のループのヘッドであり，リージョンの次のブロックの 1 つがそのリージョン自身のヘッダであることをいう．

スケジューリングの後は，リージョン拡張ブロックに対する**ブックキーピング**（プログラムの意味を維持するための命令の挿入などの操作）を行い，最後に，必要な情報（たとえば，スケジューリング結果や隣接するブロックのスケジューリングに必要なデータフロー情報）を残してリージョンを 1 つのブロックに簡約し，現在のプロシージャの CFG を更新する．

4.2 リージョン形成

リージョン形成は，リージョンとすべきブロックを選択するリージョン選択，末尾複写，および，拡張ブロックの選択よりなる．

4.2.1 リージョン選択

リージョン選択においては，まず最初に，リージョンのヘッダを選択する．基本的には，ループ・ヘッドから後続のブロックに向かって未スケジュールのブロックを探し，それをリージョンのヘッダとする．

次に，リージョン・ヘッダよりリージョンを拡張していく．リージョン・ヘッダより制御が到達する確率の高いブロックにある命令ほど移動による性能向上が期待できるので，静的分岐予測を利用し，制御が到

達する確率の高いブロックに向かってリージョンの拡張を行う．リージョン拡張において，拡張過程にあるリージョンの出口にあるブロックの1つを，今，**現在のブロック**と呼ぶこととする．現在のブロックの次のブロックは，次の条件のいずれも満たさなければ，現在のリージョンに加える．

(1) 現在のブロックがプロシージャ呼び出し命令を含む．
(2) 現在のブロックの次のブロックが動的にしか定まらない（たとえば，レジスタ間接ジャンプ命令）．
(3) 次のブロックが現在のリージョンの中にある（すなわち，リージョンがループをなしている）．
(4) 次のブロックが（別の）ループのヘッドである．
(5) 次のブロックがすでにスケジュールされている．
(6) 現在のリージョンの内部にある分岐を含むブロックの数がCCRのエントリの数と等しく，次のブロックが分岐を含む．
(7) 現在のブロックから次のブロックへの分岐確率が，あらかじめ定めた値以下である．

以上の方法でリージョンを選択した後，4.1節（図5）で説明したように，現在のループとリージョンの関係で，スケジューリング手法を切り替える．リージョンがループをなすかどうかは，リージョン・ヘッダがループのヘッドか，および，条件(1)，(2)，(5)，(6)によって決まる．また，ループ外にリージョンが広がるかどうかは，ループの終了/脱出確率と条件(7)によって決まる．すなわち，ループの繰返し確率が高いときは，ループの終了/脱出確率が小さいので，条件(7)によってループ外にある後続ブロックが選択されない．そうでなければ，この逆となる．

繰返し確率の高いループに対して行うソフトウェア・パイプライニングは，基本的には，**モジュロ・スケジューリング**[17]を用いている．ループの中でも頻繁に実行されるパスのみを選択しているので，ループ全体に対してパイプライン化するよりもコンパクトなスケジュール結果が得られる．これに関しては本論文ではこれ以上詳しく述べない．文献18)を参照してほしい．

4.2.2 末尾複写

リージョン選択後，末尾複写を行う．すなわち，リージョンにサイド・エッジがある場合，そのエッジが入るブロックより後方で，リージョン内のブロックをすべて複写し，サイド・エッジを取り除く．

4.2.3 拡張ブロックの選択

最後に，拡張ブロックを選択する．4.2.1項で述べたリージョンの選択において，条件(1)，(5)，(6)でリージョンの拡張が停止した場合，リージョンの次のブロックを拡張ブロックとする．この条件に当てはまるブロックは，コード量を著しく増加させることなく利益ある命令移動が可能と考えられるためである．

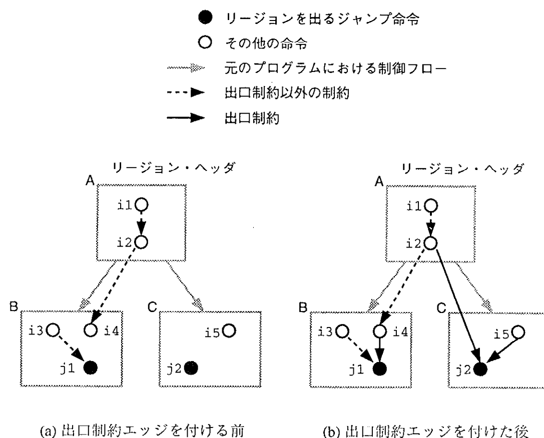


図6　出口制約
Fig. 6　Exit constraint.

4.3 制約グラフの作成

リージョンと拡張ブロック内の命令に対してデータ依存を計算し，スケジューリングの制約を表すDAGを作成する．

リージョン内の命令に対しては，さらに**出口制約**と呼ぶエッジを追加する．出口制約とは，リージョンの入口よりある出口に至る制御パス上のすべての命令が，その出口よりリージョンを出るためのジャンプ命令がスケジューリングされるサイクルか，それ以前にスケジューリングされることを保証する制約である．この制約は，依存される命令のない命令，すなわち，DAGにおいて後続命令のない命令より，その命令が属するブロックから制御が到達するすべての出口のブロック内のジャンプ命令に対して，コスト0のエッジを張ることにより実現する．

図6を用いて出口制約を説明する．同図(a)は，リージョンをABCとし，出口制約エッジを張る前のDAGである．パスAB上にある命令 `i1`～`i4` は，Bからリージョンを出るジャンプ命令 `j1` に対して出口制約がある．`i1`～`i4` の中で，パスABに含まれるDAGにおいて後続命令のない命令は `i4` である．したがって，`i4` より `j1` に対して出口制約エッジを張る．同様に，パスAC上にある命令 `i1`, `i2`, `i5` は，Cからリージョンを出るジャンプ命令 `j2` に対して出口制約がある．`i1`, `i2`, `i5` の中で，パスACに含まれるDAGにおいて後続命令のない命令は `i2`, `i5` である．したがって，`i2`, `i5` より `j2` に対して出口制約エッ

ジを張る．同図(b)に，出口制約エッジを張った後のDAGを示す．

リージョン内部の命令はすべて，出口制約によって，リージョンから制御が出る前にスケジューリングされる．一方，拡張ブロックにある命令は，必要な命令だけ限定してリージョン内に移動しスケジューリングするために，この制約は加えない．

DAGにはこのほか，既スケジュールの拡張ブロック内の命令に対し，**長命令制約**と呼ぶ制約を追加する．長命令制約エッジは，拡張ブロックにおいて，各サイクルにスケジューリングされている命令に対して，その1つ前のサイクルにスケジューリングされているすべての命令より張るエッジである．長命令制約に関しては4.5節で述べる．

4.4 リージョンのスケジューリング

トップ・ダウンにリスト・スケジューリングによって，リージョンおよび拡張ブロック内の命令のスケジューリングを行う．すなわち，DAGにおいて先行命令がなく遅延を満たした命令を，優先度の高い順にできるだけ早いサイクルにスケジューリングする．実行頻度が高く，依存パスの長い命令ほど優先してスケジューリングすべきであるから，ある命令の優先度は，DAGにおけるその命令のノードの高さ（リーフへの最大パス長）と，その命令が属するブロックにリージョン・ヘッダから制御が移行する確率の積を基本とする値とした．拡張ブロック内の命令もリージョン内の命令と同一の式で優先度を与える．

リージョン内のすべての命令がスケジューリングされれば，スケジューリングを終了する．具体的には，リージョンを出るすべてのジャンプ命令がスケジューリングされれば，スケジューリングを終了する．出口制約によりスケジューリング終了時点でリージョン内のすべての命令がスケジューリングされていることが保証される．一方，拡張ブロック内の命令に対しては出口制約がないので，優先度に従って性能向上に必要な命令のみが選択され，リージョン内に移動されスケジューリングされる．これにより出口近傍での並列度の低下を防ぐ．

スケジューリング後，必要であればブックキーピングを行う．ブックキーピングに関しては4.5.2項で述べる．

4.5 拡張ブロックからの命令移動

拡張ブロックが未スケジュールのブロックであれば，リージョンへの命令移動に関して特別な制約はない．一方，拡張ブロックがすでにスケジューリングされたブロックである場合，未スケジュールのブロックと同様にデータ依存のみを制約として移動すると，移動後，残った命令について拡張ブロックを再びスケジューリングする必要がある．なぜならば，拡張ブロックの実行サイクル数が残された命令が原因で減少しなければ，移動による性能向上がないからである．本節では，すでにスケジュールされたブロックからの命令移動について議論する．その後，拡張ブロックからの命令移動にともなうブックキーピングについて議論する．

4.5.1 すでにスケジュールされたブロックからの命令移動

既スケジュールの拡張ブロックからの命令移動による性能改善を，再スケジューリングせず実現するために，次のようにした．

- 拡張ブロックからは早いサイクルにスケジューリングされている命令ほど先に移動する．
- 移動されずに残った先頭の命令にリージョンからのジャンプ先を変更する．

つまり，拡張ブロックの最初の n サイクルの命令がすべてスケジューリングされれば，現リージョンからは，第 $(n+1)$ サイクルのスロットにジャンプするようにする．こうすれば，現リージョンから拡張ブロックを通過するパスの長さは n サイクル短くなり，拡張ブロックも再スケジューリングを行う必要がない．

このために，既スケジュールの拡張ブロック内の命令をDAGに加える際，その拡張ブロックにおいて，あるサイクルにスケジューリングされているすべての命令から，その次のサイクルにスケジューリングされているすべての命令に重み0の制約エッジを加えた．この制約を長命令制約と呼ぶ．長命令制約は，第 n サイクルの命令がすべて移動されなければ，第 $(n+1)$ サイクルの命令は移動されないことを保証する．

この制約の下では，現リージョンのスケジューリングが終了した時点では，一般的には，最初の n サイクルのすべての命令がリージョン内に移動され，第 $(n+1)$ サイクルの命令は，一部の命令だけがリージョン内に移動され，第 $(n+2)$ サイクル以降の命令は移動されていないという状態となる．移動された第 $(n+1)$ サイクルの命令は，同一サイクルに移動されずに残された命令があるために，リージョンから拡張ブロックを抜けるパスを短くすることに寄与しない．そのため，移動された第 $(n+1)$ サイクルの命令のスケジューリングを無効化し，元の拡張ブロックに戻し，リージョンからのジャンプ先を拡張ブロックの第 $(n+1)$ サイクルの最初の命令とする．

この制約の下で，もう1つ考慮しなければならない点は，リージョン内に移動された命令と拡張ブロック

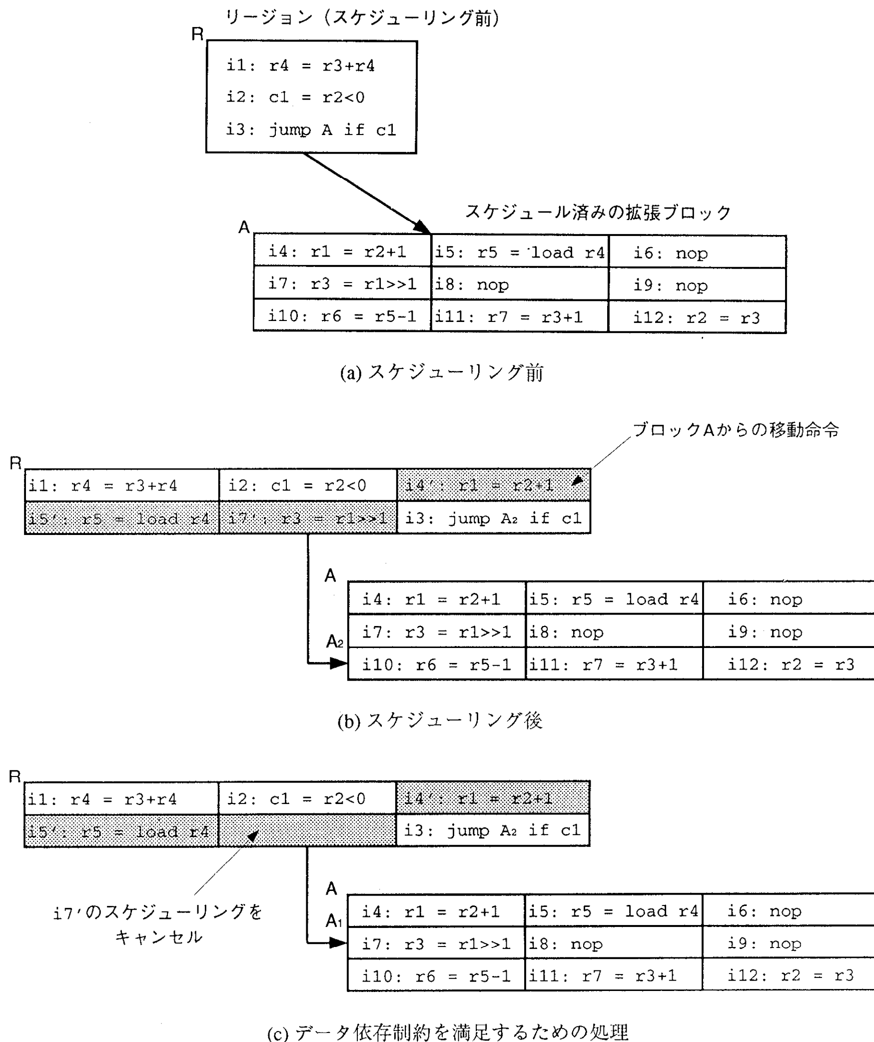


図 7 スケジュール済みの拡張ブロックからの命令移動
Fig. 7 Code motion from an already-scheduled block.

に移動されずに残された命令との間の依存関係である．命令移動時には，先行制約は考慮するが後方制約は考慮しないので，移動された命令と残された命令との間のデータ依存関係は，必ずしも満足されない．**図 7** に示す例を用いて説明する．すでにスケジューリングされたブロック A からの命令移動を考える（図 7 (a)）．図 7 (b) は，リージョンのスケジューリングが終了した時点での，リージョンのスケジュール表と，リージョンからブロック A への飛び先を表している．命令スケジューラは，ブロック A から命令 i4，i5，i7 を現リージョンに移動している（現段階では，これらの命令をコピーしている）．リージョンのスケジューリング前には，ブロック A では，命令 i5 と i10 の間のデータ依存関係は満足されていたが，命令移動後は，i5 のコピー i5' と i10 の間のデータ依存関係は満足されていない☆．

この問題に対しては，リージョンから拡張ブロックへの飛び先を，リージョン内の命令と拡張部に残された命令との間のデータ依存が満足するまで，拡張ブロックの先頭に向かって移動し，スケジューリングをキャンセルすることによって対処する．具体的には，リージョンと拡張ブロック内の命令の間のデータ依存が満足されているかどうかを検査し，満足されていない場合，現在の飛び先より拡張部のスケジュールにおいて 1 サイクルだけ戻し，そのサイクルの命令の

☆ ロード命令のレイテンシは 2 サイクル，その他の命令のレイテンシは 1 サイクルとしている．また，ハードウェアにはインターロック機構はないとしている．

リージョンへのスケジューリングをキャンセルする．これを，飛び先が拡張ブロックの先頭に至るまで繰り返す☆．図7(c)では，飛び先を1サイクル分小さなアドレスとし，命令 i7' のスケジューリングをキャンセルした．これによって，命令 i5' と i10 のデータ依存関係は満足された．

4.5.2 ブックキーピング

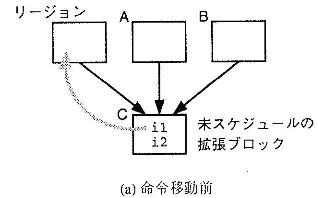
不正な投機的命令移動に対してハードウェア支援を持たない一般的なマシンでは，通常，命令移動時に**レジスタ・リネーミング**を行う．すなわち，書き込みレジスタを空のレジスタに変更し移動，後に，その変更した書き込みレジスタから元のレジスタに値をコピーする命令を挿入する．プレディケーティングでは，ハードウェアが対処するので，レジスタ・リネーミングの必要はない．ジョイン（制御の合流点）を越える移動で複写を必要とする場合は移動を行わない☆☆．末尾複写によってサイド・エッジが取り除かれているので，リージョン外への命令の複写の必要は生じない．ただし，拡張ブロックに対しては末尾複写を行わないので，ジョイン・ブロックの場合，命令移動に対して複写が必要である．拡張ブロックのスケジューリングがすでに行われているかどうかで，以下に示すように処理が異なる．

拡張ブロックが未スケジュールのジョイン・ブロックの場合

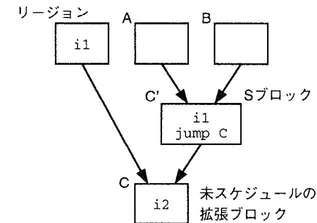
この場合，拡張ブロックから現リージョンへ命令を移動した際，リージョン以外の先行ブロックと拡張ブロックの間にブロックを作成し，そこへ移動命令のコピーを入れ，拡張ブロックからそれらの命令を削除する．**図8**に示す例では，リージョンに移動された命令は，新たに作成したブロック C' にコピーされ C より除かれる．C' には，C への制御移行を可能にするためにジャンプ命令を加える．我々は，複写された命令を入れるブロック（例では C'）のことを **S ブロック**（**split block**）と呼ぶ．この方法は，リージョンから拡張ブロックを抜けるパスの性能を向上させるが，リージョン以外のパスの性能を低下させる．たとえば，図8に示す例で，ブロック C の命令 i1, i2 を互いに依存のない独立な命令とすると，ブロック C の分離

☆ DAG 作成時に現リージョンの後方のリージョンからのデータ依存制約は，(存在するならば) DAG に加えられている（実現方法の説明は省略する)．したがって，拡張ブロックの先頭までスケジューリングをキャンセルすれば，データ依存が満足されていることが保証されている．

☆☆ ノード分割（4.7.2 項で述べる）によって，この制限による性能低下を抑える．なお，ジョインを越える命令移動でも複写を必要としない場合は移動を行う．



(a) 命令移動前



(b) 命令移動とSブロックの生成

図8 ブックキーピング
Fig. 8 Bookkeeping.

前には，ブロック C の実行は 1 サイクルしか必要としないが，ブロック C の分離後は，命令 i1 と i2 はそれぞれ別の基本ブロックに属しているので，それらが直列に実行され，その結果，性能低下が起こる．さらに，S ブロックが，同図(b)のようにジョイン・ブロックであると，さらに分離される可能性があり，その場合性能低下の可能性はさらに大きくなる．

そこで，プロファイルを用い，リージョンから拡張ブロックへ制御が移行する頻度が，他のパスを通って拡張ブロックへ制御が移る頻度に比べてある定められた値より大きい場合のみ，このブロックを拡張ブロックとすることとした．これにより，リージョン外のパスに与えるペナルティが著しい性能低下を引き起こす場合を避ける．

さらに性能低下を防ぐため，S ブロックは形成後すぐに単独でスケジューリングする．既スケジュールの拡張ブロックからの命令移動では，ブックキーピングによる新しいブロックの生成はなく（後述），S ブロックのこれ以上の分離による性能低下を防ぐ．さらに，拡張ブロック内のすべての命令が移動されれば，スケジューラはそのブロックを消去し，性能低下を防ぐ．

拡張ブロックが既スケジュールのジョイン・ブロックの場合

既スケジュールの拡張ブロックからの命令移動では，長命令制約を加えることにより，早いサイクルの命令より順に移動されるようにし，移動後はリージョンからのジャンプ先アドレスを変更することを述べた．現リージョン以外のブロックからは，拡張ブロックへのエントリ点は変わっていないので，この方法でブック

キーピングは完了する．

4.6 繰返し確率の高くないループの最適化

繰返し確率の高くないループにおいては，リージョンはループ外にも広がる．これは，ループの外からの命令移動にも利益があることを示している．この場合，ループの異なるイタレーションからのバック・エッジに沿った命令移動と，ループの外からの命令移動の両方を考慮し，バランス良くスケジューリングを行うことが性能向上につながる．

我々のアルゴリズムは，スケジューリングを繰返し試すことによって，バック・エッジに沿って移動すべき命令の集合を見つける．具体的にはまず最初に，バック・エッジに沿って移動する命令をループ・ヘッドより発見的に選択する．それらをリージョンのDAGの先頭より除き後方に追加した後，スケジューリングを行い性能を評価する．これを移動命令を変えて繰返し，最も高い性能を得る移動命令の集合を見つける．繰返しにおける各スケジューリングにおいては，リージョンのDAGはループの次のイタレーションの命令とループ外の命令を同時に含むので，これらの命令をスケジューリングすることによる利益が同時に考慮される．

この方法では，繰返し回数を少なくし，計算コストを下げる配慮が必要である．このために，本アルゴリズムは，ループ内のDAGの高さの等しい命令を，高さの高い順に移動命令の集合に加え評価する．性能向上がなくなった時点で繰返しを止める．バック・エッジに沿った命令移動はループ内DAGの高さを低くすることによって，ループをコンパクトにするので，DAGの高さの等しい命令を組として評価を繰り返すことは，妥当であると考える．

このアルゴリズムを，前節で説明した拡張ブロックからの命令移動アルゴリズムを利用して実現する．**図9**を用いて説明する．同図において，命令 i1〜i3 に付けた括弧内の数字は，ループ内DAGにおける高さである．

まず，通常のスケジューリング時と同様に，リージョン・ヘッダ（この場合，ループ・ヘッド）よりリージョンを形成する（図9(a)）．ループの繰返し確率が高くない場合なので，ループ外のブロックもリージョンに含まれる．

次に，ループのバック・エッジに沿って移動する命令を入れるための基本ブロックを，ループ・テールとループ・ヘッドの間に挿入する．このブロックを，我々は**オーバラップ・ブロック**と呼ぶ．オーバラップ・ブロックは，この時点では，単にループ・ヘッドへのジャンプ命令を含むブロックである．

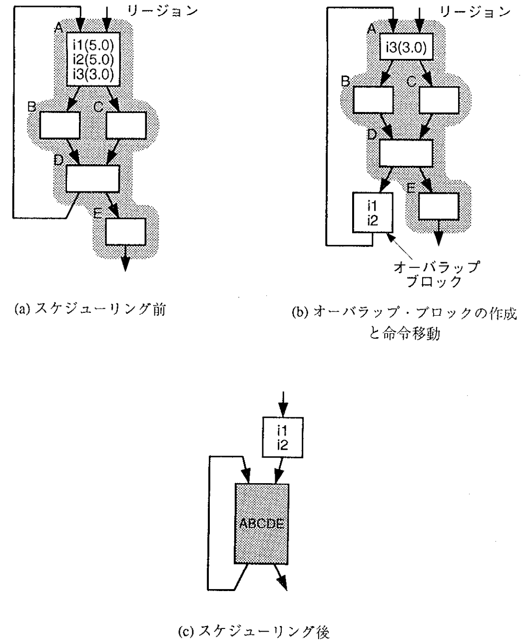


図9 繰返し確率の高くないループのスケジューリング手順
Fig. 9 Scheduling process for an infrequently-iterative loop.

次に，ループ・ヘッドよりオーバラップ・ブロックへ命令を選択し移動する．命令 i1，i2 は，DAGにおける高さが最も高く等しいため，これらの命令を移動する（図9(b)）．移動する命令は，リージョンのDAGより除き，DAGの後ろに付け加える．優先度の再計算が必要であるが，それは，付け加えた命令をリーフとするDAGの部分グラフに対してのみ行えばよいので，計算量は非常に小さく問題ない．

この後，オーバラップ・ブロックを現リージョンに対する未スケジュールの拡張ブロックとし，リージョン全体をスケジューリングし性能を評価する．スケジューリング時には，オーバラップ・ブロックからは，リージョンの性能向上に必要な命令のみ限定して移動される．

さらに繰返しが必要と判断すれば（後に説明する），移動命令を追加して同様にスケジューリングを行い性能を評価する．この例では，図9(b)において，さらに命令 i3 をオーバラップ・ブロックに移動し評価する．

この実現方法では，繰返しがこれ以上必要ないと判断する場合は2つある．

1つは，オーバラップ・ブロック内の命令はすべてリージョン内に移動されたが，前回に比べて性能の改善がなかった場合である．DAGにおける高さの高い

順に移動命令を増加させているので，この場合，これ以上移動命令を増加させても性能が向上する可能性はほとんどない．

もう1つは，リージョンのスケジューリング終了時において，リージョン内に移動されずにオーバラップ・ブロックに命令が残されている場合である．この場合，残った命令とは，データ依存制約などによりループ・バックのジャンプ命令がスケジューリングされるまでにスケジューリングできなかったか，あるいは，他の命令（たとえば，ループの外の命令）の方がより優先度が高く，スケジューリングされなかったかのいずれかである．いずれにしても，移動されずに命令が残されていることは，オーバラップ・ブロックにある命令をすべて移動することに利益がなかったことを示しており，これ以上移動命令を増加させる必要がない．

繰返しが不要であると判断した場合，前回のスケジュール結果が本アルゴリズムにおける最善の結果である．オーバラップ・ブロックより今回加えた命令をDAG の先頭に戻し，再スケジューリングを行い，その結果を最終的なスケジュール結果とする．また，オーバラップ・ブロックに移動された命令は，ループの前にブロックを作りブックキーピング・コピーを置く．図 9 (c) にスケジューリング後の CFG を示す．

4.7 その他の最適化

これまでに述べた以外で，現在の我々のスケジューラに実現されている主な最適化について述べる．

4.7.1 ループ不変命令の移動

ループ不変命令[15]とは，ループ内の命令で，制御がそのループ内にある限り実行結果の変わらない命令をいう．この命令は，プログラムの意味が変わらなければ，ループの前に移動しループ内より除去できる．この結果，ループの実行サイクル数を減少させることができる場合がある．

我々の命令スケジューラは，従来の最適化が施された後のコードを入力とする☆．したがって，命令スケジューリング以前の最適化段階で可能なループ不変命令のループ外への移動は，すでに行われている．命令スケジューリング時点において，さらにこの最適化を行うのは次の理由による．

- 我々のリージョン最適化スケジューリング手法では，ループの一部を選択し末尾複写によってサイド・エッジを取り除くため，データフローが変わる．このため，元のループで不変でない命令でも，リージョンの中では不変となる場合がある．
- ループ不変命令が制御依存を持っている場合，従来の最適化段階では，ループ外への移動を行わないが，これを行う．すなわち，ループ不変命令の投機的移動を行う．

従来のマシンで，ループ不変命令のループ外への投機的移動をレジスタ・リネーミングにより実現する方法では，リネームしたレジスタを元のレジスタにコピーする命令の挿入によって，移動の前後で命令数に変化がない．このため，ループ実行のサイクル数が減少する可能性は高くない．これに対して，プレディケーティングでは，レジスタ・リネーミングで必要とされるコピー命令は必要なく，移動により必ず命令数が減少するので，ループ不変命令移動の有効性は，移動が投機的であっても高い．また，従来のマシンでは行うことができない危険な投機的移動も，プレディケーティングでは可能なので，最適化の適用範囲が広い．

以上のように，リージョン最適化スケジューリング手法とプレディケーティングを組み合わせれば，ループ不変命令移動による最適化は，従来の場合に比べて適用する機会が多く，同時に有効性も高い．

4.7.2 ノード分割

ジョイン・ブロック内の命令が，異なる制御パス上の複数の命令にデータ依存する場合がある．この場合，依存する命令がすべてスケジューリングされ，それらの制御依存がすべて解消しなければ，その命令はスケジューリングできない．**図 10** を用いて説明する．今，命令 **i1** は逐次的に，命令 **i3** は投機的にすでに実行されているとする．命令 **i4** のオペランド **r5** は，命令 **i1** か **i3** のどちらかの実行結果である．したがって，命令 **i4** の実行開始時には，どちらであるか定まっている必要がある．このためには，命令 **i3** の実行結果がコミットまたは無効化されていなければならない（コミットならば **i3** の結果，無効化ならば **i1** の結果）．これにより，命令 **i4** のスケジューリングは，命令 **i3** の制御依存が解消されるサイクル以降と制限さ

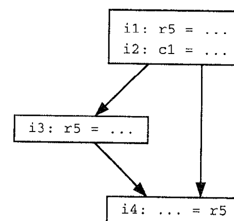


図 10　コミット依存
Fig. 10　Commit dependence.

☆ MIPS のコンパイラにより最適化したコードを入力としている．

れる．

このスケジューリング制約は，データ依存する命令がコミット，または，無効化されることにより解消するので，我々は**コミット制約**[10]，あるいは，**コミット依存**と呼ぶ．図10の例では，命令 i4 は，命令 i2 の実行により分岐条件 c1 が定義され，その結果，命令 i3 の実行結果がコミットまたは無効化するまでスケジューリングできない．このため，DAG において命令 i2 から命令 i4 にコミット制約エッジを張る．

この制約は，ジョイン・ブロックを複写し，そのブロックへの制御パスを単一にすれば解消できる．これを**ノード分割**[7]と呼ぶ．ノード分割はコード量の増加をともなうので，我々は，ジョイン・ブロックへ入る制御の頻度に大きな偏りがある場合のみノード分割を行う．これによって，実行頻度の低いパスに沿ったデータ依存により生じるコミット制約で，ジョイン・ブロック内の命令のスケジューリングが遅れることを防ぐことができる．

5. 性能評価

本章では，提案のスケジューリング手法の有効性について評価する．評価は，投機的実行に対する支援能力を段階的に強化した4つのマシン・モデルに対して行う．最初に性能評価の方法を述べ，次に評価したマシン・モデルについて説明する．各マシン・モデルに対して，特有のスケジューリング制約と技術が必要なので，それについても説明する．最後に評価結果を示す．

5.1 評価方法

比較における基本とするスカラ・マシンを MIPS R3000[19]とし，R3000のサイクル数を評価するVLIWマシンの実行サイクル数で割った値を性能向上とした．VLIW マシンの実行サイクル数は，トレース駆動シミュレータを作成して測定した．我々のシミュレータは，スカラ・マシンの命令トレースより，VLIW マシン向けにスケジューリングしたコードのどこを制御が移行しているかを判断し，ハードウェアの動的な振舞いをシミュレートする．シミュレータが VLIW マシンのコードにおける制御移行を正しく判断できるように，命令スケジューリングの際に行った CFG の変換に関する情報を，シミュレータに与える．

5.2 評価モデル

評価したモデルのいずれも，4つの ALU，4つの分岐ユニット，2つのロード・ユニット，1つのストア・ユニット，4つの CCR エントリ，2つのデータ・キャッシュ・ポートを持ち，4命令を同時に実行する．命令のレイテンシは R3000 の命令のレイテンシとほぼ等しく，ロード命令のレイテンシが2サイクルで，その他の整数演算命令のレイテンシは1サイクルである．2ビットの分岐履歴を持つ2K エントリの**分岐先バッファ**[20]を用いて分岐予測を行う．予測ミス時のペナルティは1サイクルとした．また，キャッシュ・ミスはないと仮定した．

評価を行った4つのモデルを以下に示す．

- トレース・スケジューリング・モデル（T モデル）

基本的に，従来の単純な VLIW マシンである．ただし，パイプライン無効化によって投機的命令移動を支援する．命令スケジューラはトレースを選択し，その中で命令移動を行う．コンパイラは，危険な命令移動を行う場合，パイプライン内で無効化できる範囲に限定してスケジューリングする．不正な投機的命令移動に対しては，レジスタ・リネーミングを行う．レジスタ・リネーミングによってコピー命令が生成されるが，可能であれば**コピー伝搬**[15]によって解消する．さらに，コピー伝搬の結果，コピー命令の実行結果が不要となれば，これを消去する[15]．これによって，レジスタ・リネーミングにおけるコピー命令挿入によるペナルティを削減する．

- non-excepting トレース・スケジューリング・モデル（T-N モデル）

T モデルの機能に加えて，投機的例外への従来の解決機構である non-excepting 命令方式[13),21]を備えるとする．**図11**を用いて non-excepting 命令方式を説明する．non-excepting 命令方式では，通常の命令と同一の操作を行うが，例外処理を起動しない non-excepting 命令と呼ぶ命令を用意する．危険な命令移動を行う場合は，non-excepting 命令をスケジュール

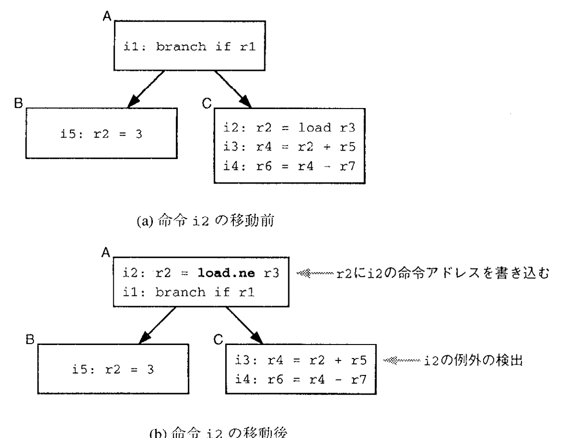


図11　non-excepting 命令による投機的例外の検出
Fig. 11　Detection of a speculative exception caused by a non-excepting instruction.

する．図11の例で，命令 i2 を分岐命令 i1 の上方に移動する場合，i2 を non-excepting ロード命令に変えて移動する．同図(b)の命令 i2 の添え字 .ne は，そのロード命令がnon-excepting 命令であることを示す．もしも，non-excepting 命令が例外を起こした場合は，例外処理を行なわず，書き込み先（例では r2）にその命令のアドレスを書き込み，さらに，例外を起こした命令の実行結果であることを示すマークを付ける．後に，他の命令（例では i3）がマークされたデータを参照した際に，例外処理を行う．

この方式は，投機的例外の処理を延期させることができるため，危険な命令移動が可能である．ただし，マシン状態再構築の機構がないので，危険な投機的命令に依存する命令を投機的移動することはできない．図11の例では，命令 i3，i4 の投機的移動はできない．我々の命令スケジューラは，non-excepting 命令をスケジューリングした場合，その命令の結果を参照する命令の投機的な移動を制限する．すなわち，パイプライン無効化可能な範囲までしか移動しない．さらに，non-excepting 命令の結果を参照する命令が，移動元のブロックに存在しない場合，live なレジスタ数を増加させないために☆，スケジューラは結果を参照する副作用のない命令☆☆を挿入し，例外発生を検査する．この場合も，この検査用命令の投機的移動を制限する．

non-excepting 命令方式実現のために，ハードウェアの追加が必要である．この内，主要なものは，レジスタ・ファイルの各エントリに設ける例外発生を示すマークを記憶するための1ビットのレジスタである．明らかにこのハードウェア量は非常に小さい．したがって，T モデルに対してハードウェア量はほとんど増加しないということができる．

- non-excepting プレディケート・モデル（P-N モデル）

T-N モデルの機能に加えて，命令にプレディケートを付加し，ハードウェアは従来のプレディケート実行[7),12),13)]を行うとする．すなわち，命令にプレディケートを付加し，分岐の両側のパスからの命令移動を支援する．non-excepting 命令方式における命令移動制限は存在する．ハードウェア量は，T-N モデルと同様の理由で，T モデルに比べてほとんど増加しない．

- プレディケーティング・モデル（P モデル）

プレディケーティング機構をハードウェアが持ち，命令スケジューラは，投機的命令移動を自由に行うことができるとする．このモデルでは，プレディケーティングによりハードウェア量が増加する．主要な増加は，図2に示したレジスタ・ファイルによる．文献11)によれば，8つの読み出しと4つの書込みポートを持つ32エントリの通常のレジスタ・ファイルに必要なトランジスタ数は，約31 K であるのに対して，同一のポート数とエントリ数のプレディケーティング（CCR のエントリ数4）向けのレジスタ・ファイルに必要なトランジスタ数は，約66 K である．したがって，約35 K の増加となるが，この量は，最近のハイエンドのマイクロプロセッサを実現するために必要なトランジスタ数（数M）[22)]のわずか数%であり，多くない．

5.3 評価結果

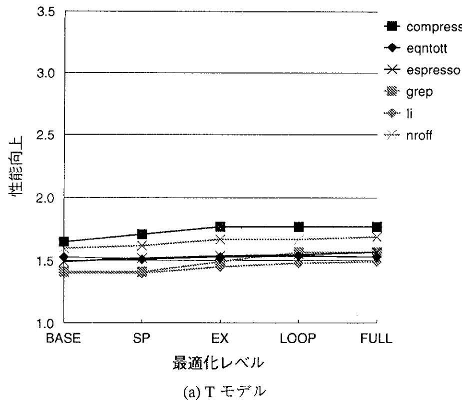
前節で述べた各モデルについて，次に示す5つの命令スケジューリングの最適化レベルで評価を行なった．

- BASE
  4.2節で述べた方法でループ内よりリージョンを形成し，4.3節，4.4節で述べた方法でリージョン内の命令をスケジューリングする．拡張ブロックからの命令移動，リージョンの先頭からの命令移動，ノード分割，ループ不変命令のループ外への移動は行わない．
- SP
  BASE に加えて，リージョンがループをなしていれば，ソフトウェア・パイプライニング[18)]を用いて最適化する．これを「従来の最適化」と呼ぶこととする．
- EX
  SP に加えて，拡張ブロックからの命令移動を行う．
- LOOP
  EX に加えて，ループの繰返し確率に合わせた最適化を行う．すなわち，繰返し確率が高い場合は，ソフトウェア・パイプライニングを適用し，繰返し確率が高くない場合は，4.6節で提案した最適化を行う．
- FULL
  LOOP に加えて，4.7節で述べたノード分割とループ不変命令の投機的移動を行う．SP に加えて行う最適化を「リージョン最適化」と呼ぶこととする．

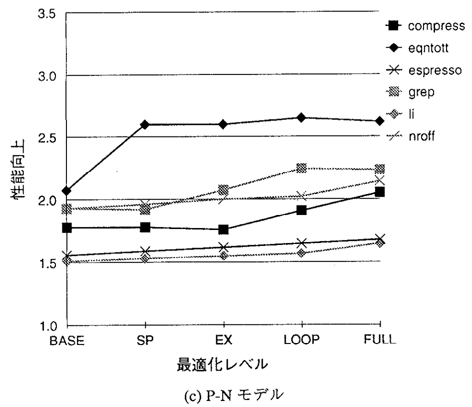
図12に各最適化レベルでの性能測定結果を示す．compress（データの圧縮），eqntott（真理値表の生

☆ 例外命令の参照オペランドは例外検出時点まで保存されなければならない．したがって，例外検出が遅れると live なレジスタが増加し，レジスタ割当てが圧迫される．

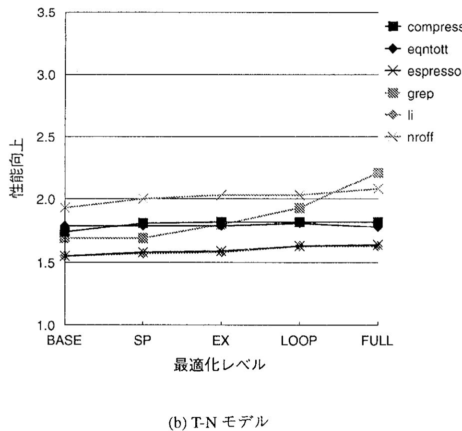
☆☆ 現在のインプリメンテーションでは，結果を参照し r0 に書き込みを行う命令としている．r0 は値ゼロの読み出し専用レジスタであり，副作用は生じない．



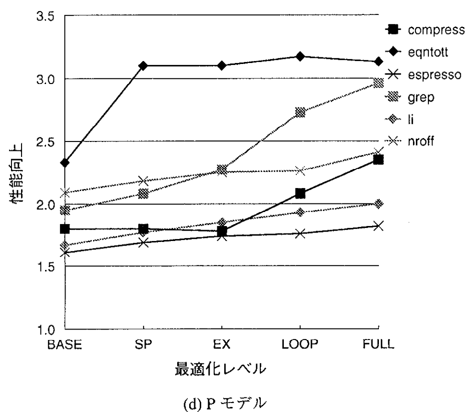
(a) T モデル



(c) P-N モデル



(b) T-N モデル



(d) P モデル

図 **12**　性能向上
Fig. 12　Speedup.

成)，espresso (PLA の最適化)，li (Lisp の翻訳) は SPEC ベンチマーク・プログラム，grep (文字列の検索)，nroff (文書の清書) は UNIX のユーティリティである．これらはすべて非数値計算応用プログラムである．

また，**図 13** に，従来の最適化とリージョン最適化による性能向上をまとめる．棒グラフの下の部分は，従来の最適化による性能向上であり，その上の部分はリージョン最適化による性能向上である．

従来の最適化に対してリージョン最適化により，各マシン・モデルにおいて，最大で 11.3% (T モデル) ～42.3% (P モデル)，幾何平均で 5.3% (T モデル) ～16.5% (P モデル) の性能向上が得られた．命令移動制限の小さいモデルほど性能改善率が大きく，最適化が有効であることが分かる．最適化の項目を個別に見れば，P モデルで，拡張ブロックからの命令移動によって平均で 2.9%，繰返し確率が高くないループの最適化によってさらに 7.3%，ノード分割とループ不変命令の移動によってさらに 6.3%性能が向上した．

図 13 から分かるように，compress，grep で最適化

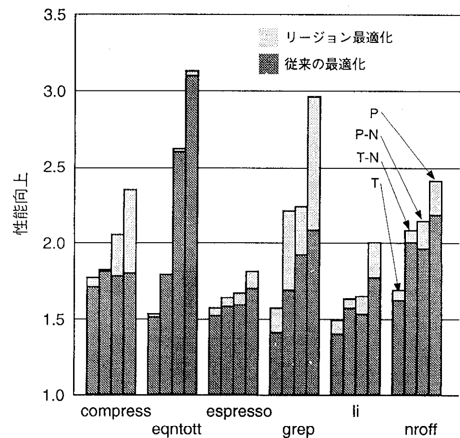


図 **13**　リージョン最適化による性能向上
Fig. 13　Speedup through region optimizations.

による性能改善が著しい．これらのプログラムには，実行頻度は高いが繰返し確率の高くないループがあり，これに対する最適化の効果が顕著に現れた．compress では，さらにノード分割により性能を向上させること

表1 命令スケジューリング時間の増加
Table 1 Increase of instruction scheduling time.

モデル	従来の最適化	リージョン最適化
T	1.05	1.11
T-N	1.08	1.15
P-N	1.05	1.09
P	1.00	1.04

ができた．また，grep では，ループ不変命令をループの外に投機的移動することにより，ループをコンパクトにスケジューリングできたため，さらに大きな性能向上を達成することができた．

逆に，eqntott では，最適化によってほとんど性能改善はなかった．eqntott では，繰返し確率の非常に高いただ1つのループが実行サイクルのほとんどを消費する．このループにリージョン最適化を適用できる部分がなかったためである．

P モデルのスカラ・マシンに対する最終的な性能向上は，2.40 倍に達した．この大きな性能向上の達成は，コンパイラの最適化とハードウェアの投機的実行支援の協力によるものである．特に，ハードウェアの投機的実行の支援による効果は大きく，T モデルに対して 50.0%性能を改善することができた．5.2 節で述べたように，プレディケーティングによるハードウェアの増加量は，プロセッサ全体に必要なハードウェア量のわずか数%なので，この大きな性能改善を得るためのコストとしては十分に小さい．

これに対して，non-excepting 命令による投機的実行への支援は，ハードウェアの増加量は非常に小さいが，不十分であり，性能を大きく改善することができず，T-N モデルは T モデルに対して 15.6%性能を改善するだけであった．従来のプレディケート実行機能を加えた P-N モデルは，分岐予測の困難なプログラムの性能を改善することができるが，27.5%の改善にとどまった．

表1 に，ハードウェア・モデルや最適化の違いによる命令スケジューリング時間の増加を示す．P モデルに対し従来の最適化を行った場合を基準とし，各場合における全ベンチマーク・プログラムのスケジューリング時間の合計を，基準の場合のスケジューリング時間で割った値で表している．スケジューリング時間は，MIPS のコンパイラで最適化したアセンブリ・コードを読み込み，スケジュール結果を出力するまでの時間である．表1において,「従来の最適化」の欄は，最適化レベル SP まで最適化した場合の時間に対する値であり,「リージョン最適化」の欄は，最適化レベル FULL まで最適化した場合の時間に対する値である．MIPS のコンパイラで最適化したアセンブリ・コードを入力としているので，ソース・コードを入力とした場合の全コンパイル時間の増加率は表1に示した値より小さい．なお，時間の計測は，主記憶 64 M バイトの Sun SPARCstation 2（CPU: Weitek SPARC Power *P, 32.2 SPECint, 31.1 SPECfp）で行った．基準の場合のスケジューリング時間は，154.8 秒であった．

表1に示すように，P モデルにおいて，リージョン最適化によるスケジューリング時間増加率は，わずか 4%と非常に小さく問題ない．P モデル以外では，レジスタ・リネーミング，コピー伝搬，non-excepting 命令向けの制御が余分に必要なので，従来の最適化だけでもスケジューリング時間が 5～8%増加する．リージョン最適化を行えば，T-N モデルで最大 15%まで増加するが，T-N モデルの従来の最適化に対する時間増加率は 6%（1.15/1.08）と小さく問題ない．また，T-N モデルでのスケジューリング時間が，P-N モデルでのスケジューリング時間より，従来の最適化で 3%（1.08/1.05），リージョン最適化で 6%（1.15/1.09）長い．これは，T-N モデルでは命令ウィンドウとしてトレースをとるので，リージョンをとる P-N モデルより，プログラム全体を細かく分解しスケジューリングすることとなり，その結果，スケジューリング時間が増加している．

6. まとめ

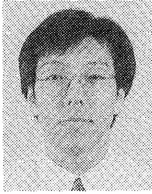
本論文では，非数値計算応用に適した VLIW マシンのための広域命令スケジューリング手法を提案した．特に，命令ウィンドウの出口近くでの性能低下の緩和と，繰返し確率の高くないループに対するスケジューリングに対して，有効な手法を示した．また，ループ不変命令のループ外への投機的移動など，本スケジューリング手法のフレームワークとプレディケーティングに適した最適化手法を示し，その有効性を確認した．評価の結果，我々の手法を用いれば，プレディケーティングを備えたマシンにおいて，従来の手法に対して大幅に性能を改善できることを確認した．また，最適化の効果は，投機的実行に対するハードウェアの支援能力によって大きく影響を受け，その重要性が定量的に確認された．提案のスケジューリング手法とプレディケーティングを組み合わせることによって，スカラ・マシンに対して 2.40 倍の性能向上を達成することができることが分かった．

謝辞 本研究に対してご支援いただいたシステム LSI 開発研究所部長・角正氏に感謝致します．

参 考 文 献

1) Wall, D.W.: Limits of Instruction-level Parallelism, *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.272–282 (1991).
2) Lam, M.S. and Wilson, R.P.: Limits of Control Flow on Parallelism, *Proc. 19th Int. Symp. on Computer Architecture*, pp.46–57 (1992).
3) Tokoro, M., Tamura, E. and Takizuka, T.: Optimization of Microprograms, *IEEE Trans. Comput.*, Vol.C-30, No.7, pp.491–504 (1981).
4) Nicolau, A.: Percolation Scheduling: A Parallel Compilation Technique, Computer Sciences Technical Report, 85-678, Cornel University (1985).
5) Fisher, J.A.: Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Trans. Comput.*, Vol.C-30, No.7, pp.478–490 (1981).
6) Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J. and Hwu, W.W.: IMPACT: An Architectural Framework for Multiple-instruction-issue Processors, *Proc. 18th Int. Symp. on Computer Architecture*, pp.266–275 (1991).
7) Mahlke, S.A., Lin D.C., Chen, W.Y., Hank, R.E. and Bringmann, R.A.: Effective Compiler Support for Predicated Execution Using the Hyperblock, *Proc. MICRO-25*, pp.45–54 (1992).
8) 小松秀昭，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大域的コードスケジューリング技法，情報処理学会論文誌，Vol.37, No.6, pp.1149–1161 (1996).
9) Hwu, W.W., et al.: The Superblock: An Effective Technique for VLIW and Superscalar Compilation, *The Journal of Supercomputing*, Vol.7, pp.229–248 (1993).
10) Ando, H., Nakanishi, C., Hara, T. and Nakaya, M.: Unconstrained Speculative Execution with Predicated State Buffering, *Proc. 22nd Int. Symp. on Computer Architecture*, pp.126–137 (1995).
11) 安藤秀樹，中西知嘉子，原　哲也，中屋雅夫：プレディケーティング：VLIW マシンにおける投機的実行のためのアーキテクチャ上の支援，情報処理学会論文誌，Vol.37, No.11, pp.2039–2055 (1996).
12) Hsu, P.Y.T. and Davidson, E.S.: Highly Concurrent Scalar Processing, *Proc. 13th Int. Symp. on Computer Architecture*, pp.386–395 (1986).
13) 小松秀昭，古関　聰，鈴木秀俊，深澤良彰：拡張VLIW プロセッサ GIFT における命令レベル並列処理機構，情報処理学会論文誌，Vol.34, No.12, pp.2599–2610 (1993).
14) Ebcioglu, K. and Nicolau, A.: A Global Resource-constrained Parallelization Technique, *Proc. 3rd Int. Conf. on Supercomputing*, pp.154–163 (1989).
15) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison-Weslay, Reading, MA (1986).
16) Allen, J.R., Kennedy, K., Porterfield, C. and Warren, J.: Conversion of Control Dependence to Data Dependence, *Proc. 10th ACM Symposium on Principles of Programming Languages*, pp.177–189 (1983).
17) Lam, M.S.: Software Pipelining: An Effective Scheduling Technique for VLIW Machines, *Proc. ACM SIGPLAN'88 Conf. on Programming Language Design and Implementation*, pp.318–328 (1988).
18) 中西知嘉子，安藤秀樹，原　哲也，中屋雅夫：パス選択によるソフトウェア・パイプライニング，情報処理学会研究会報告，95-HPC-57, pp.127–132 (1995).
19) Kane, G.: *MIPS RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ (1988).
20) Lee, J.K.F. and Smith, A.J.: Branch Prediction Strategies and Branch Target Buffer Design, *Computer*, Vol.17, No.1, pp.6–22 (1984).
21) Colwel, R.P., Nix R.P., O'Donnell, J.J., Papworth D.B. and Rodman, P.K.: A VLIW Architecture for a Trace Scheduling Compiler, *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp.180l–192 (1987).
22) 浅見直樹，枝　洋樹：次世代マイクロプロセサ，スーパスカラと VLIW が融合，日経エレクトロニクス，No.626, pp.67–150 (1995).

(平成 8 年 9 月 10 日受付)
(平成 9 年 2 月 5 日採録)

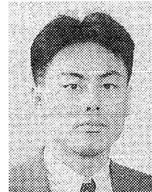


安藤 秀樹（正会員）

1959年生．1981年大阪大学工学部電子工学科卒業．1983年同大学大学院修士課程修了．同年三菱電機（株）LSI研究所入社．ISDN用ディジタル信号処理LSI，第5世代コンピュータ・プロジェクトの推論マシン用プロセッサの設計に従事．1991年Stanford大学に客員研究員として留学．1996年京都大学工学博士．1997年名古屋大学大学院工学研究科電子情報専攻・講師．計算機アーキテクチャ，コンパイラの研究に従事．

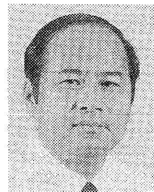
中西知嘉子

1988年大阪大学基礎工学部情報工学科卒業．同年三菱電機（株）LSI研究所に入社．以来，計算機用LSIの開発に従事．現在，同社システムLSI開発研究所所属．



原 哲也（正会員）

1966年生．1989年九州大学工学部情報工学科卒業．1991年同大学大学院総合理工学研究科情報システム学専攻修士課程修了．同年三菱電機（株）LSI研究所に入社し，細粒度並列処理アーキテクチャの研究に従事．1996年より信号処理プロセッサの開発に従事．現在，同社マイコン・ASIC事業統括部所属．



中屋 雅夫（正会員）

1951年生．1974年早稲田大学理工学部電子通信学科卒業．1976年同大学大学院修士課程修了．1988年工学博士（早稲田大学）．1976年三菱電機（株）入社．以来，高速MOSゲートアレイ，ECLゲートアレイ，MOSA/D，D/Aコンバータ，三次元回路素子，通信用LSI，ISDN用LSI，並列処理プロセッサ，ATM-LAN用LSIなどのシステムLSIの研究開発に従事．現在，三菱電機（株）システムLSI開発部勤務．1993年度電子情報通信学会論文賞受賞．IEEE，電子情報通信学会各会員．