命令レベル並列計算機上で並列実行する領域の選択を高速に行う方法

田端 邦男[†] 小松 秀昭[†]

プレディケート付き命令を許す命令レベル並列計算機においては，条件分岐を削除して分岐先命令群に相補的なプレディケートを付けたり（IF 変換），制御依存やメモリ依存を超えた投機的な命令移動によってクリティカルパスを縮めたりすることで，プログラムの効率的な実行が実現できる．しかし，これらの最適化の適用領域（以降ハイパーブロックと呼ぶ）によっては，分岐先命令のクリティカルパス長のバランスの悪さ，実行可能性が低い命令の挿入などの原因で，実行効率改善の程度が異なり，実行効率の低下の可能性すらあるため，ハイパーブロックとする領域の適切な選択が必要である．従来の研究は，さまざまなハイパーブロックの選び方に対して実際にコードスケジュールして効率を比較するという，Generate & Test 型の方法であり，Test されるハイパーブロックの場合の数は，分岐命令の数に従って指数関数的に増大するため，広範囲に高い実行効率を得ようとすると，コンパイル時間が多くなってしまう問題があった．本研究の基本アイデアは，プログラムのある領域全体（a とする）をハイパーブロックとしたときの実行効率を，その領域を構成する部分（b，c など）の実行効率から再帰的に見積もれるとモデル化することにある．このモデルに動的計画法を適用することで，(a) の効率が (b), (c) を逐次に実行した場合より悪くなる場合を検出し，(b) と (c) を独立なハイパーブロックとする判断が高速に終了する．これらの仕組みは，基本ブロックの数にほぼ比例する時間で終了し，なおかつ，コードスケジューラに直接依存しないため，高速かつ広範囲に対し，高い実行効率を実現する．

A Fast Method to Select Hyperblock on Instruction Level Parallel Processors

Kunio Tabata[†] and Hideaki Komatsu[†]

For the efficient execution on instruction-level parallel processors with predicated instructions, some optimization techniques are essential, such as converting a conditional branch into complementary predicated instructions (IF-conversion), control speculation, and data speculation. The performance improvement achieved by these techniques varies depending on where to apply and not to apply the optimization because of the insertion of infrequently executed code, or the imbalance of the critical paths in branch targets. Compilers are required to select suitable program regions (Hyper Block) for these optimizations. To satisfy the requirement, existing studies propose a Generate-&-Test-style method, in which a hyper block selector forms all the possible Hyper Blocks, queries the estimated execution time driving code schedulers, and chooses a case of Hyper block selections with the best efficiency. Since the execution time of the method increases exponentially according to the number of the branch, the method spends impractically large compilation time for wide regions in programs. We propose a faster and efficient method even when the Hyper Block candidate area is wide. The key idea of our method is that a hyper block selector estimates the approximately best execution time of a region based on the subsets of the region recursively. The recursion starts from code-scheduled Basic Blocks, and stops when estimated execution time is worse than the sequential execution. This idea is implemented by dynamic programming, which achieves almost linear compilation time, and allows this method to be applied to wide areas in programs.

1. はじめに

近年，プログラムの実行速度の高速化手法として，命令レベル並列実行が注目されており，それを可能にするものの 1 つとして Very Long Instruction Word（VLIW）を実装したプロセッサが開発され，実用化されようとしている．これに対応した言語処理系には，この機能を生かした最適化が要求され，特に，プログラムから適切に並列性を抽出し，実行効率の向上を実現する方法が研究されている．本研究は，並列実行する領域の適切な選択の新たな方法を提案する．

† 日本アイ・ビー・エム株式会社東京基礎研究所
IBM Research, Tokyo Research Laboratory

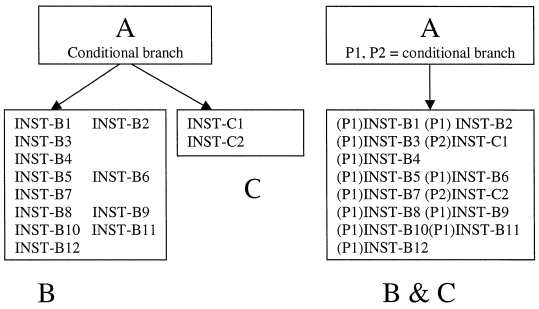


図 1 IF 変換すると実行効率が悪くなる例：INST-BN, INST-CN は，それぞれ，基本ブロック B と基本ブロック C の中の命令を表す．並列している命令は並列に実行されるとする．

Fig. 1 An example of an unsuitable case of IF-conversion: INST-BN and INST-CN are instructions in basic block B and basic block C. The instructions which are put horizontally represents that they will be executed in parallel.

本研究が対象とする，プレディケート付き命令を持つ VLIW 計算機では，条件分岐命令を削除して分岐先命令群に相補的なプレディケートを付け（以降これを「IF 変換」と呼ぶ），並列に実行させることで，プログラムの並列性を高められる．しかし，IF 変換は，分岐先命令のクリティカルパス長のバランスの悪さや実行可能性が低い命令の挿入などの原因で，必ずしも実行効率を高めるとは限らない．

たとえば，**図 1** の左図のように，分岐先（B と C）の実行効率が非常に偏ったものに対して IF 変換を適用し，図 1 の右図のようにプレディケート付き命令群に変換した場合の効率を考える．仮に，B と C への分岐確率が等しいとすると，図 1 の左図は，8 サイクルと 2 サイクルの平均 5 サイクルで実行され，図 1 の右図は，どちらの分岐条件を満たしても 8 サイクルで実行される．このような場合，明らかに，IF 変換を適用しないほうが実行効率が高い☆．

よって，IF 変換を適用すべき分岐命令を適切に選択する必要がある．すべての条件分岐命令に IF 変換を適用する領域を，本研究では「ハイパーブロックの候補☆☆」と呼び，この領域を適切に決めることが必要となる．

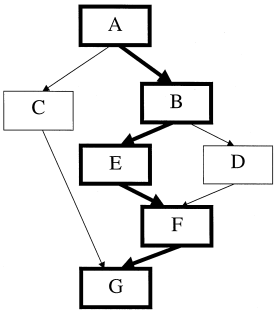


図 2 制御フローの例：A, B, E, F, G が最頻度で実行される経路であり，C と D を並列に実行するかどうか判定が必要である．

Fig. 2 An example of a control flow: A, B, E, F, G is the hottest path, and it is required to decide C & D is included in a hyperblock.

これを実現する非常に素直な方法としては，あらゆるハイパーブロックの候補の選び方それぞれについて，実際にコードスケジュールをして，その実行効率が最良のものを選ぶという方法が提案されている．たとえば，**図 2** では基本ブロック A と B の分岐命令について，(1) A も B も IF 変換しない，(2) A は IF 変換し B は IF 変換しない，(3) A は IF 変換せず B は IF 変換する，(4) A も B も IF 変換する，の各場合についてそれぞれコードスケジュールして結果を比較し，最良のものを選択することになる．この方法は，理想的なコードスケジューラがあるという仮定のもとで理想的な実行効率が実現できる一方，分岐命令の数に従って指数関数的に組合せが増加し，現実的な時間で終了しない可能性がある．また，仮になんらかの方法により探索空間を狭めたとしても，コードスケジューラを起動することから大きな計算時間を要し，実践的とはいえない．さらに，コードスケジューラにより厳密な実行効率を高速に見積もること自体が，非常に難しい課題である．

また，分岐確率などの実行時情報が得られる場合は，最も実行頻度が高い経路の実行効率を犠牲にしないように，ハイパーブロックの候補を選択するという方法も考えられる．たとえば，図 2 の太線矩形と太線矢印で示した部分を最頻度実行経路とすると，C と D をハイパーブロックの候補とするのは，最頻度実行経路の実行効率を犠牲にしない場合のみである．この方法は，最頻度実行経路の最適化のみを優先すれば十分であるといえるが，それ以外の部分の効率を考慮すると課題が残る．それは，1 つの基本ブロックに対する判定が，他の基本ブロックに対する判定に影響を与えうることである．たとえば，図 2 の D をハイパーブロッ

☆ 実際には，右図では分岐命令が削除されることと，それにともない分岐予測の失敗ペナルティを回避できるため，命令サイクル数をもう少し削減できる．

☆☆ 厳密には，ハイパーブロックとは，制御フローの流入がただ 1 つで，流出が複数あるような連続した基本ブロックの集合に対して，IF 変換を適用したものである．ここでは，制御フローの条件は考慮されていないため，「ハイパーブロックの候補」とする．「ハイパーブロックの候補」は，実際には複数の「ハイパーブロック」に分割される場合がある．たとえば，図 2 で「ハイパーブロックの候補」が，A, B, E, F, G とすると「ハイパーブロック」は A, B, E と，F と G とに 3 つに分割される．

クの候補とするか判断したいとする．Cがハイパーブロックの候補でないときには，Dはハイパーブロックに含めてよいがCをハイパーブロックの候補にする場合は，Dはハイパーブロックに含めない方がよい場合は起こりうる．このことは，前述の組合せ爆発と同様な計算量の増大の可能性を示している．よって，このような方法は，精度の高い実行時情報が得られ，かつ，実行経路が最頻度実行経路に非常に偏っている場合以外では効果を得るのは難しく，ある特定のアプリケーションプログラム以外で実践的に効果を得るのは難しい．

本研究が解決する課題は，「ハイパーブロックの候補」を，実践的な言語処理系に耐えうる高速な方法で，なおかつ最頻度実行経路のみではない広い範囲に対して，高効率な実行を実現するように決定することである．

以降，2章では本研究の概要を述べつつ貢献を明らかにする．続いて，3章で関連研究について述べ，4章でアルゴリズムの概要と適用例を示し，5章で評価結果を述べる．最後に6章で結論と今後の展望を述べる．

2. 本研究の貢献

本研究の基本アイデアは，プログラム全体を階層的に表現し，親の階層の実行効率を，子の階層（親の階層を構成する領域）の実行効率から，再帰的に見積もれるようにモデル化することである．こうすることで，動的計画法が適用できるようになり，プログラムの全体の実行効率の見積りが終了した時点で，適切なハイパーブロックの選択が決定できる．さらに，動的計画法を用いているので，モデル化が現実に則している限りにおいては，最適なハイパーブロックの選択が可能である．

以降，実行効率の見積りを可能にするモデル化に注目してその概要を述べる．具体的には，「階層的表現の方法」と「子の階層から親の階層の効率を見積もる仕組み」を順に述べる．

本方法の階層構造は，基本ブロックの接続の形態によって構成される．具体的には，プログラム全体から，並列に実行可能な基本ブロックのペアと，逐次的にのみ実行可能な基本ブロックのペアを検出し，さらに，それらペアどうしでも，並列に実行可能なペアと逐次に実行されるペアを検出し，階層的に接続させる．

並列に実行可能な基本ブロックのペアを，実際に並列に実行する場合の効率は，前述のように実際にコードスケジュールしてみないと分からないが，本研究の手法は，この部分で以下のようなアイデアを用いることで高速動作する．

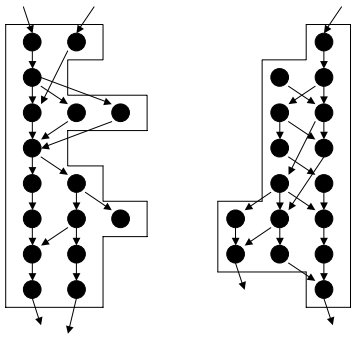


図3 基本ブロックの実際の内容
Fig. 3 Contents of basic block.

一般に，実際のプログラムの並列性は，プログラムの個所によってさまざまに変化する．たとえば，**図3**は左右2つの基本ブロックを表し，その中の黒丸が1つの命令を表していて，黒丸を結ぶ矢印はデータ依存を表しているのだが，これが示すように，プログラムの並列性は基本ブロックによって異なり，基本ブロックの中でも依存関係によってさまざまに変化する．

本研究の手法は，ここに，平均並列度という考え方を取り入れる．まず，基本ブロックの内部のデータ依存関係を解析し，クリティカルパス長と，延べ計算時間（逐次実行したときの所要時間）を求めておく．延べ計算時間をクリティカルパス長で割ったものを平均並列度とする．たとえば，図3では，左右とも平均並列度は2である．

いま，図3の左右の基本ブロックを並列に実行しようとしたとき，その並列度は，基本ブロックの平均並列度の和であるとする．別の見方をすれば，左右の基本ブロックの凹凸が相殺される形でコードスケジュールが可能であるという見通しを持って，実行効率を見積もる，といえる．この見通しは，基本ブロックのみならず，複数のハイパーブロックから1つのハイパーブロックを再帰的に構成するときも同様に満たされると見なす．

このように，本研究の方法は，このモデルが現実のプログラムの性質と近い限りにおいて，高速かつ高精度で並列実行の効率を見積もることが可能で，この仕組みを利用することで，広い範囲においてハイパーブロックの候補の適切な選択を高速に行える．

この特徴は，特に，本研究が対象とするJava Just In Time Compilerにとって非常に有用である．それは，Just In Time Compilerのような動的コンパイラは計算時間が非常に限られ，さらに，Javaのような汎用アプリケーションを扱う言語は，最頻度実行経路

に実行が極端に集中するという仮定を置くのは合理的ではないからである．

3. 関連研究

文献 1) は，どの領域を並列実行し，IF 変換を実行することで性能向上が得られるかという課題について，発見的手法による 1 つの解を与えている．まず，最も実行可能性が高いと予想される実行パス（main trace）を特定し，そこは無条件で並列実行することにする．続いて，それ以外のパス（sub trace）それぞれについて，main trace と同じ並列実行領域に含めるかどうか判定し，段階的に並列実行領域を増大させてゆく．その結果，main trace の高速化を最優先しつつ，それを阻害しない程度に sub trace を含んだ領域が並列実行される．具体的に，ある分岐命令に対し，IF 変換を実行するかどうかの判定は，以下の 4 つの要因が考慮される．

- sub trace に pipeline を乱す命令があるかどうか
- main trace に対する sub trace の実行確率
- main trace に対する sub trace の機械語命令数の比
- ハードウェアの並列実行能力の限界

この方法は，main trace が実際に高い頻度で実行される場合に，高い効率を実現する．また，コンパイル時には，main trace の分岐命令の数を n とすると，n に比例する程度の計算時間を要する．さらに，この方法は，Hammock 型条件分岐命令に対し，分岐先のバランスが非常に悪い場合には code duplication を実行して cotrol flow のマージを削除し，実行効率の悪化を防いでいるが，その定量的な評価はされていない．

文献 2) は，まず，プログラム全体を 1 つの並列実行領域とし，IF 変換を行い，それに，さまざまな最適化を施し，後に選択的に IF 変換の逆を実行することで，結果的に分岐命令を選択的に IF 変換した状態を作る方法である．この方法は，コードスケジューラと協業して，各分岐命令について，逆 IF 変換した場合，しない場合それぞれについて，実行サイクル数を求めて，どちらが実行性能が高いかによって逆 IF 変換するかしないかを決定する．ただし，この方法をプログラム内のすべての分岐命令に対して適用すると，組合せ爆発が起こり，現実的な時間で終了しない．このため，リストスケジューラと協業して，クリティカルパスをスケジュールする際に逆 IF 変換を試みることで，計算量を抑えている．具体的には，クリティカルパス上の分岐命令の数 n に対し，2n 回程度スケジューリングを行う手法を提案している．

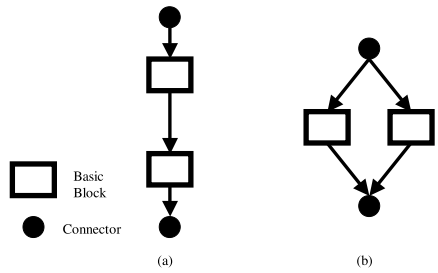


図 4 シリーズパラレルグラフの構成要素.（a）シリーズ接続,（b）パラレル接続
Fig. 4 Elements of series parallel graph, (a) series connection, (b) parallel connection.

4. アルゴリズム

4.1 概　　要

本研究の方法を示す準備として，まず，プログラム全体をハイパーブロックとした場合の実行効率を見積もる方法を述べ，のちに，それに付け加える形で，ハイパーブロックの候補の選択の方法を示す．

4.1.1 実行効率を見積もる方法

実行効率を見積もる基本アイデアは，プログラム全体を階層的に表現し，親の階層の実行効率を，子の階層（親の階層を構成する領域）の実行効率から，再帰的に見積もれるようにモデル化することである．以降，「どのように階層的に表現するか」，「どのように，子の階層の実行効率から，親の階層の実行効率を見積もるか」について順に述べる．

まず，ハイパーブロックの候補となりうる領域全体の制御フロー☆を，基本ブロックの逐次接続と並列接続のみの組合せによって構成されるシリーズパラレルグラフに変換する．**図 4** (a) がシリーズ接続を表し，図 4 (b) がパラレル接続を表す．始点と終点を共有するシリーズ接続のみによって構成される領域をシリーズスート，始点と終点を共有するパラレル接続のみによって構成される領域をパラレルスートと呼ぶことにすると，領域全体は，シリーズスートとパラレルスートのネスト関係のみによって表現できる．

次に，基本ブロックを，縦がクリティカルパス長，横が平均並列度であるような矩形領域と見なす．これは，縦がクリティカルパス長を下回らない限り変形可能である．たとえば，**図 5** (a) の基本ブロックを，(b), (c) のように縦に伸ばす形で変形可能である．見方を

☆ ここでは，説明の簡略化のため制御フローとしたが，実際は後述の詳細な説明では依存グラフを用いる．これにより，プログラムの並列性をより引き出すことができるが，基本的には制御フローに対して適用することも可能である．

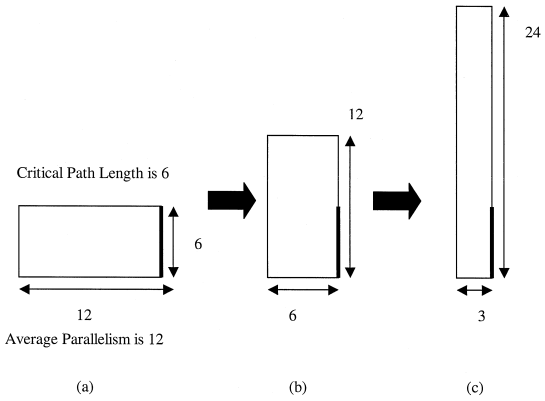


図 5 基本ブロックのモデル化
Fig. 5 Modeled basic block.

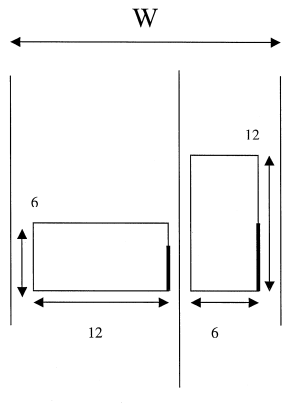


図 6 パラレルスートのモデル化
Fig. 6 Modeled parallel suite.

変えると，

規則 1：任意の基本ブロックに平均並列度 w を与えると所要実行時間が見積もれる

といえる．ちなみに，プロセッサの並列度を W として，1 回の所要時間見積りが 1 単位時間で行われるとすると，$1 < w < W$ のすべての w の整数値に対して所要実行時間を見積もる操作は，W 単位時間以下で終了する．

続いて，**図 6** の基本ブロック 2 つから構成されるパラレルスートの実行効率の見積りを例にとる．2 つの基本ブロックの境界線 X を左右に動かしながら，両者の縦の長さの最大が最小になる点を見つける．この作業を，並列度を示す図 6 の W を変化させながら繰り返す．これにより，基本ブロックのみから構成されるシリーズスートについて，「平均並列度 w を与えると所要実行時間が見積もれる」という規則 1 と同様な事項が成り立つ．ちなみに，この部分に要する計算時間は，プロセッサの並列度を W とし，2 つの基本ブロックの縦の長さから大きい方を選択する操作を 1 単位時間で行うとすると，W^2 単位時間以下である．

さらに，基本ブロック 2 つから構成されるシリーズスートの実行効率は，各並列度に対する実行効率を単に加えることで求められるとすると，同様に，基本ブロックのみから構成されるシリーズスートについて，「平均並列度 w を与えると所要実行時間が見積もれる」．ちなみに，この部分に要する計算時間は，プロセッサの並列度を W とし，加算操作を 1 単位時間で行うとすると，W 単位時間以下である．

以上より，パラレルスート，シリーズスートの構成要素が基本ブロック以外の場合でも，再帰的に「平均並列度 w を与えると所要実行時間が見積もれる」のであるから，まとめると，

規則 2：任意のスートに平均並列度 w を与えると所要実行時間が見積もれる

といえる．

4.1.2 ハイパーブロックの候補を選択する方法

ここまでで，任意のスートにおける任意の平均並列度に対する，実行時間の見積り方法を示した．ここに，本研究の本来の目的である，ハイパーブロックの候補を決定する仕組みを付け加える．

IF 変換が実行効率を悪くする例を前述したが，このような場合を正しく検出し，同一のハイパーブロックを構成しないようにするというのが基本アイデアである．具体的には，IF 変換によって実行効率が悪くなる可能性があるのは，パラレルスートをハイパーブロックとする場合であるから，パラレルスートの実行効率見積りの際にこれを検出する．

パラレルスートの実行効率見積りフェーズでは，各構成要素の任意の並列度 w に対する実行時間の見積りがあらかじめ分かっている．よって，これらの値と，明示的な分岐命令生成によるオーバヘッドなどを考慮することで，ハイパーブロックの候補とすべきか否かが判定できる．たとえば，あるパラレルスートを並列度 w で実行する際の効率を見積もるとすると，前述の方法で，パラレルスートをハイパーブロックとした場合の効率と，明示的な分岐により実行した場合の効率，すなわち，構成要素それぞれの並列度 w での効率に分岐命令によるペナルティを加えた値を比較し，最良の場合を選択する．

ただし，この判定は，このパラレルスートの並列度によって異なる可能性があり，最終的な判断は，このパラレルスートが最終的にいくらの並列度で計算されるかによる．

最後に，これらの仕組みの実行に要する計算時間を述べる．すでに述べてきたように，プロセッサの並列度を W とすると，基本ブロックに対する操作は W 単位時間が，パラレルスートに対する操作は W^2 単位時間が，シリーズスートに対する操作は W 単位時間が必要である．スートの総数は基本ブロックの総数を超えることはないので，基本ブロックの総数を n とすると，$n \times W^2$ に比例する計算時間で終了する．現実的には，ハードウェアの並列度 W は 10 程度かそれ以下の小さい定数であることから，基本ブロックの数に比例する時間で終了する．

4.2 手続きの実際

前節で述べたアイデアの概要を実現するアルゴリズムを手続き的に述べる．

準備の段階

(1) **適用領域の決定**

実行頻度情報やプログラムの構造を基に，並列実行する領域の候補を決める．たとえば，ループの中にあり，ループを含まない領域や，他の最適化の過程で生まれた低頻度実行領域を含まない領域が該当する．また，IBM JDK1.3 が持つ選択的コンパイルの仕組みと，インタプリタモードから，コンパイルモードへの途中遷移の仕組みを利用して獲得した実行時情報に基づいて，ある程度以上の実行頻度があるとされる部分も候補とする．

(2) **Program Dependence Graph [3] の作成**

(1) の領域内の Basic Block ごとに，クリティカルパス長と延べ計算時間の見積りを求めておく．ある基本ブロックをまたぐような制御フローを検出し，そこに，クリティカルパス長 0，延べ計算時間 0 のダミーノードを作る．これを含めた制御フローと基本ブロックに対し Program Dependence Graph（PDG）をつくる．また，基本ブロックの内部は，IR 表現の 1 命令ごとに，データ依存と副作用による依存を求め，そのクリティカルパス長を求めておく．また，各命令を逐次に実行した場合の所要時間を，その基本ブロックの延べ計算時間とし，それをクリティカルパス長で割った値を平均並列度とする．

(3) **PDG の変形**

PDG ノード間の冗長な依存を削除する．たとえばノード A とノード B の間に，データ依存と制御依存が存在する場合は，集約して 1 つの依存とする．これにより，ノード間の依存は，縮退して単なる先行制約となる．

(4) **シリーズパラレルスートの抽出**

(2) でできた graph から，シリーズパラレルスートを抽出する．実際には 2 でできるグラフは完全なシリーズパラレルグラフではないので，厳密にはグラフの論理的な解析，変換が必要となる．ここでは，高速に動作する発見的手法を用い，変換を行う．さらに，シリーズパラレルスートのネスト関係を表すシリーズパラレルネスト木を生成する．この木のノードを構成するのは，基本ブロック，シリーズスート，パラレルスートの 3 種であり，この木の葉は必ず基本ブロックである．

効率見積りの段階

(5) **各スートの実行時間の見積り**

それぞれのスートについて，ハードウェアの最大の並列度で実行した場合の実行時間を見積もり，さらに，並列度を少しずつ下げながら，最後には逐次実行についても実行時間を見積もる．具体的には，対象のスートの種類によって以下のように見積りの方法が異なる．

(a) **基本ブロックの場合**
この基本ブロックのクリティカルパスの実行時間を下回らないようにしつつ，延べ計算時間を並列度で割った値を実行時間見積りとする．

(b) **シリーズスートの場合**
構成要素のそれぞれに，ハードウェアの最大並列度を超えないように，同じ並列度を与え，あらかじめ見積もられている値の和をこのスートの実行時間の見積りとする．

(c) **パラレルスートの場合**
構成要素のそれぞれに，合計がハードウェアの最大並列度を超えないように並列度を与え，あらかじめ見積もられている各要素の実行時間の最大値を求める．また，この値と，構成要素それぞれを独立したハイパーブロックとして明示的な分岐命令を生成した場合とも効率を比較し，最良のものを，このスートの実行時間の見積りとする．後に確認するための準備として，それぞれの最良の場合の構成を再現できるように記録しておく．

上記のように，あるスートの実行時間見積りを得るには，その構成要素の実行時間見積りがあらかじめ必要である．この条件は，シリーズパラレルネスト木を post order の深さ優先で走査することで満たされる．

ハイパーブロック構成の段階

(6) **結果の確認とハイパーブロックの構成**

ここまでで，各スートについて，各並列度に対する，所要実行時間の見積りが求まっている．最後に，シリーズパラレルネスト木の根を構成するスートを，ハードウェアの並列度で実行した場合について，各スートがそれぞれ独立したハイパーブロックの候補を構成する

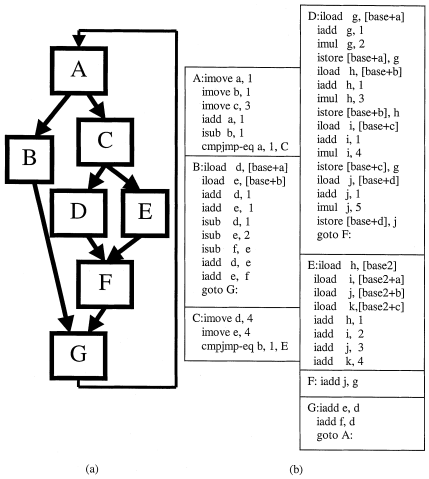


図 7 対象とするプログラムの例.(a) オリジナルの制御フローグラフ,(b) 仮想的なアセンブラコード

Fig. 7 An example program segment for hyperblock selection algorithm, (a) original control flow graph, (b) original assembly code.

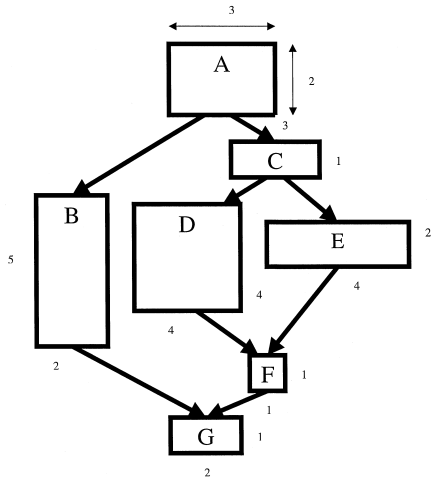


図 8 Program Dependence Graph を生成後，基本ブロック単位にクリティカルパスと平均並列度を求めた後の状態

Fig. 8 After Program Dependence Graph generation, critical path length and avarage parallelism are also estimated.

かどうかの情報を集め，ハイパーブロックの候補を決定する．ハイパーブロックの内部の条件分岐には，IF 変換を適用し，ハイパーブロックをまたぐ条件分岐には，IF 変換を適用せず，明示的な制御フローのまま残す．

なお，IF 変換のアルゴリズムは，RK アルゴリズム[4]を基にしたものを用いる．

4.3 適 用 例

図 7 に示したプログラムの例に対し，本研究の方法を実際に適用する過程を示す．まず，方法の (1) を用いて，ハイパーブロックの候補となりうる範囲を限定する．この例では，ループ内部すべてがその範囲になる．

次に，(2) に従い，PDG を基本ブロック単位の依存を基に生成し，(3) の変更を行い，先行制約グラフを生成する．基本ブロックの内部は，IR 表現の 1 命令ごとに，データ依存と副作用による依存を求め，そのクリティカルパス長を求めておく．また，各命令を逐次に実行した場合の所要時間を，その基本ブロックの延べ計算時間とし，それをクリティカルパス長で割った値を平均並列度とする．それらを基に，基本ブロックを，縦にクリティカルパス長，横に平均並列度を持つ矩形領域と見なした状態を**図 8** に示す．

続いて，(4) の手順により，シリーズパラレルネスト木を作成する（**図 9**）．太実線矩形がパラレルスートを，太破線矩形がシリーズスート，細実線矩形が基本ブロックを，それぞれ表す．たとえば，パラレルスート 2 は，基本ブロック B とシリーズスート 3 から構

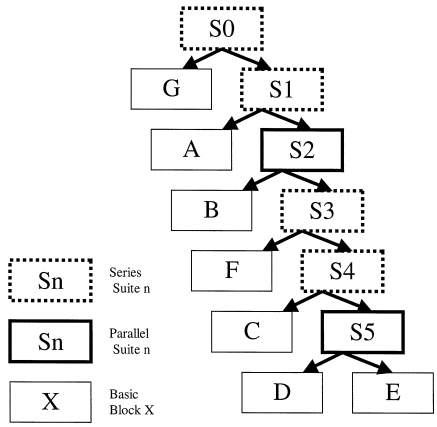


図 9 シリーズパラレルネスト木の例

Fig. 9 An example of Series Parallel Nest Tree.

成され，そのシリーズスート 3 は，最終的には基本ブロック F，C，D，E から構成される領域を表す．

続いて，(5) に従って実行時間の見積りを行う．図 9 で示したシリーズパラレルネスト木を深さ優先探索の post order でたどるため，走査順序は，D，E，S5，C，S4，F，S3，B，S2，A，S1，G，S0 の順である．

各々の基本ブロックとスートにおける，最大並列度が 6 の場合の，各並列度における実行時間の見積りを**表 1** に示した．また，各スートと各並列度の組合せにおける，ハイパーブロックの構成を**表 2** に示す．たとえば，スート $S5$ を並列度 6 で実行したときの見積り（以降 $S5(6)$ と表記する）は，表 1 より 4 サイクルだが，これを実現する構成は，表 2 より，$D(4)+E(2)$

表 1 実行時間の見積り結果
Table 1 Estimation of execution performance.

Suite\Parallelism	1	2	3	4	5	6
D	16	8	5.3	4	4	4
E	8	4	2.6	2	2	2
S5	13s	7.5s	5.45s	4.5s	4.5s	4
C	3	1.5	1	1	1	1
S4	16	9	6.45	5.5	5.5	5
F	1	1	1	1	1	1
S3	17	10	7.45	6.5	6.5	6
B	10	5	5	5	5	5
S2	13.5s	9s	7.7s	7.25s	7.25s	6.5
A	6	3	2	2	2	2
S1	19.5	12	9.7	9.25	9.25	8.5
G	2	1	1	1	1	1
S0	21.5	13	10.7	10.25	10.25	9.5

表 2 実行時間見積りの際の構成情報
Table 2 Structure information on dependency of execution performance estimation.

Suite\Parallelism	1	2	3	4	5	6
D	*	*	*	*	*	*
E	*	*	*	*	*	*
S5	D(1), E(1)	D(2), E(2)	D(3), E(3)	D(4), E(4)	D(5), E(5)	D(4)+E(2)
C	*	*	*	*	*	*
S4	C(1)+S5(1)	C(2)+S5(2)	C(3)+S5(3)	C(4)+S5(4)	C(5)+S5(5)	C(6)+S5(6)
F	*	*	*	*	*	*
S3	F(1)+S4(1)	F(2)+S4(2)	F(3)+S4(3)	F(4)+S4(4)	F(5)+S4(5)	F(6)+S4(6)
B	*	*	*	*	*	*
S2	B(1),S3(1)	B(2),S3(2)	B(3),S3(3)	B(4),S3(4)	B(5),S3(5)	B(2)+S3(4)
A	*	*	*	*	*	*
S1	A(1)+S2(1)	A(2)+S2(2)	A(3)+S2(3)	A(4)+S2(4)	A(5)+S2(5)	A(6)+S2(6)
G	*	*	*	*	*	*
S0	G(1)+S1(1)	G(2)+S1(2)	G(3)+S1(3)	G(4)+S1(4)	G(5)+S1(5)	G(6)+S1(6)

であり，これは，基本ブロック D を平均並列度 4 で，基本ブロック E を平均並列度 2 で，同一のハイパーブロックとすることである．ちなみに，$D(6)$，$E(6)$ とあれば，E(6) を独立したハイパーブロックとすることを意味し，同時に表 1 の見積り時間に s を付す．

以下の 2 つのケースについて，実行時間見積りの手順を示した．なお，スート X における並列度 n の実行時間の見積りを $X(n)$ と略記し，IF 変換しない場合の分岐のペナルティ B_p を 3 サイクルとする．

- **基本ブロック D**
  クリティカルパス長が 4 なので，並列度をいくら増やしても 4 を下回らないようにパフォーマンスが見積もられる．よって，$D(4) = D(5) = D(6) = 4$ で，その他は，延べ計算量を並列度で割った値となる．
- **パラレルスート S5**
  $S5(6)$ は，$D(4)$ と $E(2)$ を並列に実行した値が最小のため，見積り実行時間とする．たとえば，D と E を並列に実行しない場合は，$D(6)$ と $(E(6) + B_p)$ の平均値となり，4.5 で $S5(6)$ より大きいため，除外された☆．$S5(6)$ は $D(4)$ と $E(2)$ を並列に実行したとして見積もられたという情報を表 2 の $S5(6)$ 欄に「$D(4)+E(2)$」と表記する．

結局，最終結果を得るためには，$S0(6)$ から順に Sn を簡約化しながら表 2 をたどることになる．たとえば，

$$\begin{aligned} S0(6) &= G(6) + S1(6) \\ &= G(6) + A(6) + S2(6) \\ &= G(6) + A(6) + B(2) + S3(4) \\ &= G(6) + A(6) + B(2) + F(4) + S4(4) \\ &= G(6) + A(6) + B(2) + F(4) + C(4) + S5(4) \\ &= G(6)+A(6)+B(2)+F(4)+C(4)+D(4), E(4) \end{aligned}$$

となり，$E(4)$ が独立したハイパーブロックとなり，$E(4)$ 以外すべてが 1 つの異なるハイパーブロックとなる．

5. 評　　価

5.1 対象となる処理系の概要

IBM JDK 1.3 システム[5]に変更を加えたものが，本研究の評価に用いる言語処理系である．具体的には，IBM JDK 1.3 システムで使われる，アーキティクチャ透過な中間表現（以降 IR と呼ぶ）に対し，プレディケート付き命令を導入したものが最適化の対象となる．

5.2 評価の方法

IR 表現上の各命令に対し，その実行に要する時間を見積もる．各命令の所要時間は，インテルイタニアムプロセッサの機械語の実行所要時間の参考値[6]を用いる．また，仮想的なプロセッサとして，6 個の機械語命令を並列に実行でき，実際の VLIW 計算機が持つような計算資源の制約は考慮しないものを想定する．各命令間のデータ依存，制御依存と副作用による依存を求め，これらを基に IR 表現上でコードスケジューリングし，見積もられた実行時間を評価値とする．

また，本研究のアルゴリズムのほかに，上述の関連研究 1) を基にした方法の適用結果をあわせて示す．文献 1) の方法は，本来は，バランスの悪いコントロールフローに対し Node Splitting を実行し，空命令の実行によるペナルティを削減している．しかし，この方法は，文献 1) の本論でも述べられているとおり，並列度が大変高いプロセッサでないと効果的でなく，本研究の評価で用いるモデルはこれに該当しない．よって，この方法の代わりに，バランスの悪いコントロールフローはハイパーブロックから除き別のハイパーブロックにすることで，該当する分岐命令のみ，明示的な条件分岐で実行させるという方法を採用する．

さらに，この方法は，ハイパーブロック選択に先立って，その候補になりうる範囲を，ある程度以上の実行頻度が望める部分に限定する．実行頻度の推定は，実

☆ ここでは，説明の単純化のため平均値としたが，実際にはインタプリタの実行時情報などを基にした実行確率予測値により重み付けをする．

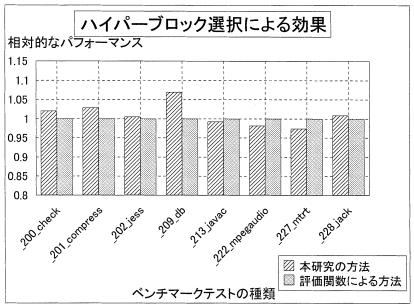


図 10 パフォーマンスの評価
Fig. 10 Evaluation of performance.

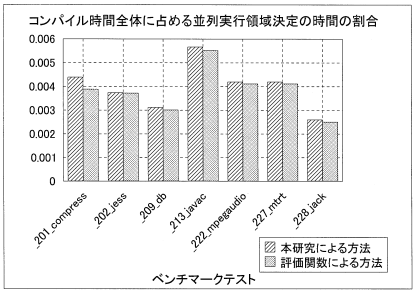


図 11 コンパイル時間の評価
Fig. 11 Evaluation of compilation time.

行時情報のほかに，言語処理系や言語仕様に依存する部分があるため，この特定の方法は，オリジナルの方法ではなく本研究と同様なものを用いることとする．

ベンチマークテストとして，SpecJVM98 [7]を用いた．IBM JDK 1.3 の持つ選択的コンパイルの仕組みを利用して，動的に実行頻度が高いと判明したメソッドのみについて，コンパイルを実行する．

5.3 結　　果

本研究のアルゴリズムを，前述の文献 1) のアルゴリズムを基にした方法に対して比較したグラフを**図 10** に示す．ベンチマークテストは SpecJVM98 [7]を用いた．本研究の方法が，従来の方法と比べ同等かもしくはそれ以上の実行効率を実現した．特に，_209_db では，従来提案されていた方法に比べ 4〜5%上回る効果を発揮していることが示され，総合的にも従来の方法をやや上回る効果を発揮していることが確かめられた．

一方，本研究の方法は，データ依存のみで構成されている，そもそも IF 変換の機会が少ないようなプログラムに対しては無力である．このようなプログラムに対しては，総当たり的に IF 変換の組合せをすべて試すような方法や，あるいはこれに適したチューニングを施した発見的手法が有効である．本研究の方法では _227_mtrt がパフォーマンスを落としているが，適用領域を概観する限りでは，IF 変換の機会が少なく，なおかつそれに対する判断が悪影響を与えている部分が散見される．よって，個々の IF 変換に対する判断の精度も高めることにより改善されると思われる．

また，**図 11** に，並列実行領域選択の仕組みがコンパイル全体に占める時間の割合の評価を示した．従来研究に比べ，ごくわずかなオーバヘッドで本研究の方法が動作したことが示された．

6. 結論と展望

本研究は，比較的広い領域からハイパーブロックとする領域を高速に選び出す方法を提案した．従来の研究と異なり，本研究は，動的コンパイラにも実装可能な高速動作が実現可能である．また，実験を通して，従来の方法に比べて同等かもしくはそれ以上の実行効率を持つことが確かめられた．今後は，VLIW プロセッサによるプラットフォームの実用化にあわせて，実機上での効率評価を行う予定である．

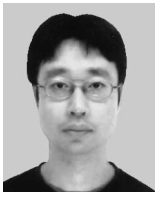
謝辞 日本アイ・ビー・エム株式会社東京基礎研究所，ネットワークコンピューティングプラットフォームの中谷登志男マネージャーをはじめ先輩社員のみなさんと，評価のベースに使った IBM JDK の開発に携わっているすべての方々に，つつしんで感謝の意を表します．

参 考 文 献

1) 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 '92*, ACM, Addison-Wesley (1992).
2) August, D.I., Hwu, W.W. and Mahlke, S.A.: A framework for balancing control flow and prediction, *Proc. MICRO '97*, pp.92–103, ACM, Addison-Wesley (1997).
3) Ferrante, J., Ottenstein, K.J. and Warren, J.D.: The Program Dependence Graph and Its Use in Optimization, *ACM Trans. Prog. Lang. Syst.*, Vol.9, No.3, pp.319–349 (1987).
4) Park, J.C. and Schlansker, M.S.: On predicated execution, HPL-91-58, Technical Report, Hewlett Packard Laboratories, Palo Alto, CA (1991).
5) Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H. and Nakatani, T.: Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal*, Vol.39, No.1, pp.287–302 (2000).
6) Intel Corporation: *Itanium Processor Microarchitecture Reference for Software Opti-*

mization (2000).
7) Standard Performance Evaluation Corp.: *SPEC JVM98 Benchmarks* (2000).
http://www.spec.org/osg/jvm98/

(平成 13 年 5 月 31 日受付)
(平成 13 年 7 月 31 日採録)



田端　邦男（正会員）
1974 年生．1999 年東京大学大学院理学系研究科情報科学専攻修了．同年日本 IBM 東京基礎研究所入社．並列処理，コンパイラの研究に従事．修士（情報科学）．

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 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）．