依存グラフを用いてアーキテクチャ独立な最適化と対象計算機の資源制約を調整する手法

稲 垣 達 氏[†]　古 関　聰[†]　小 松 秀 昭[†]

コンパイラにおける共通部分式削除やコピー伝搬などのアーキテクチャ独立な最適化と，レジスタ割付やコードスケジューリングなどのアーキテクチャ依存の最適化は，それぞれ独立したヒューリスティクスに基づいた，異なるパスとして実現されている．実際には共通部分式削除やコピー伝搬も，対象アーキテクチャの物理レジスタ数制約に対してトレードオフを持っている．しかし，これらの問題を完全に協調的に解くことは非常に計算コストが高く，現実的ではない．本研究では，アーキテクチャ独立な最適化とアーキテクチャ依存最適化の間に，物理レジスタ数の制約によるトレードオフを緩和するパスを設けることで，アーキテクチャ独立な最適化およびアーキテクチャ依存な最適化の効果を上げる手法を示す．本方式は，基本ブロック内の中間コードを演算子の依存グラフで表現し，使用レジスタ数を見積もりながら 1 パスの 4 つ組生成を行う．使用レジスタ数が少ない部分では，コードスケジューリングに対して不必要な制約を導入しないように，クリティカルパス上の演算子が連続するような 4 つ組生成を行う．使用レジスタ数が多い部分では，レジスタ割付時に余分なスピルコードを発生させないように，一時変数を減らすような順番で 4 つ組を生成する．

Using Dependence-graph-based Code Transformation to Adjust Architecture Independent Optimizations to Machine Resource Constraint

Tatsushi Inagaki,[†] Akira Koseki[†] and Hideaki Komatsu[†]

In current compilers, architecture independent code optimizations, such as common subexpression elimination and copy propagation, and architecture dependent optimizations, such as register allocation and code scheduling, are implemented as separate passes that have mutually independent heuristics. Actually, common subexpression and copy propagation also have some tradeoff against the number of physical registers of the target architecture. However, cooperative solution of these problems requires high computation cost. In this paper, we solve this problem by inserting an additional pass between architecture independent code optimizations and architecture dependent optimizations to alleviate the constraint of the number of physical registers and improve results of both optimizations. Operations in a basic block are represented by DAG (directed acyclic graph). We apply one-pass quadruple generation algorithm that considers the number of currently used registers. When register pressure is low, operations on the critical path are continuously generated not to introduce unnecessary constraint for code scheduler. When register pressure is high, we prioritize operations that reduce the number of temporary variables so that register allocator does not generate unnecessary spill codes.

1. はじめに

共通部分式の削除やコピー伝搬のようなアーキテクチャ独立な最適化と，レジスタ割付やコードスケジューリングのようなアーキテクチャに依存した機械語コードの最適化は，ともに実行プログラムの高速化を目的としている．アーキテクチャ独立な最適化の多くは，PRE (partial redundancy elimination)[14]などのデータフローベースの最適化を効率的に実装するため，プログラム中の演算子の 4 つ組表現による中間コード上の操作として実装される．

しかし，アーキテクチャ独立な中間コード最適化とアーキテクチャ依存の最適化の間にはプロセッサの資源制約，特に使用可能な物理レジスタ数によるトレードオフが存在する．すなわち，共通部分式の削除やコピー伝搬などの最適化は，プログラム中の演算コストを下げる一方，一時変数の生存区間を長くしたり，参

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

照の数を増やしたりしてレジスタプレッシャー，すなわち同時に使用されうるレジスタの個数を増やしてしまう可能性がある．中間コード最適化によって，レジスタプレッシャーが使用可能な物理レジスタの数を超えてしまうと，レジスタ割付の過程でレジスタのスピルコードが発生し，最適化の効果が相殺されてしまう．この問題に対しては，資源制約を考慮した中間コード最適化を行う手法（たとえば，レジスタプレッシャーを考慮した共通部分式削除[6]など）が考えられる．しかし，ある最適化を抑止する場合，最適化の間にはトレードオフだけでなく，依存関係も存在するので，他の最適化を妨げる可能性がある．また，レジスタプレッシャーの見積りを正確にするためには，一時変数の数を増やす可能性のある他の最適化も，前処理として行うか同様にレジスタプレッシャーを見積もって行う必要がある．

また，レジスタ割付とコードスケジューリングの間にも使用可能な物理レジスタ数によるトレードオフがある．コードスケジューリングがレジスタ割付の後のパスの最適化として実装されている場合，レジスタ割付によって生じた物理レジスタの出力依存や逆依存が命令レベルの並列性を妨げる可能性がある．たとえば，本来並列に実行できる 2 つの演算の書き込み先が，同じレジスタに割り当てられた場合，命令の発行順に制約が生じる．コードスケジューリングがレジスタ割付の前のパスにある場合は，やはりコードスケジューラーによる命令レベルの並列性抽出がレジスタプレッシャーを高くしてしまう可能性がある．この問題を解決するには，レジスタ割付とコードスケジューリング最適化を協調的に動かす仕組みが必要である[15),16)]．しかし，この方法は元々大きな計算コストを必要とするレジスタ割付とコードスケジューリングを同時に行うため，多くのコンパイル時間を必要とする．

中間コードのレジスタプレッシャーを下げる方法の 1 つとして，アーキテクチャ依存最適化の前に，演算子の依存関係の木や DAG（directed acyclic graph）による表現から，使用レジスタ数が少なくなるように 4 つ組の中間コードを生成するという手法がある[2),17)]．レジスタ割付で与えられたコード列の順序を変更したり，コードスケジューリングでレジスタ割付を変更したりするコストは大きいので，中間コードの変形でこれらのトレードオフが解消できれば，この方針は既存の最適化と親和性が高く，ポータビリティも良い．しかし，入力が共通部分式を含む場合，最適なコード生成を得る問題は NP 完全であることが知られている[1)]．したがって，Just-In-Time コンパイラのようにコンパイル時間が実行時間に含まれてしまうシステムで使用するためには，低いコストで良い解を与えるヒューリスティクスアルゴリズムの開発が重要になる．

本稿における我々のアプローチは，アーキテクチャ独立な最適化とアーキテクチャ依存最適化の間で，プロセッサの物理レジスタ数による資源制約を考慮したスケジューリングアルゴリズムを用いて，基本ブロック内で DAG からの 4 つ組中間コードの生成を行うというものである．DAG から 1 パスのコードスケジューリングを行う過程で，生成された 4 つ組が使用するレジスタの数を計算し，レジスタプレッシャーの高低によって動的にスケジューリングポリシーを選択する．レジスタプレッシャーが高い場合は，使用しているレジスタを解放する演算子を優先して，4 つ組コードの生成を行う．これによって，レジスタ割付で不要なスピルコードを生成することを避けることができる．レジスタプレッシャーが低い場合は，DAG の最もサイクル数が多くかかる経路（critical path）上にある演算子を優先した 4 つ組コード生成を行う．critical path 上の命令を連続させることは，固定サイズのウィンドウ内で最適化を行うコードスケジューリングに対して良い入力を与える．このようなパスを設けることにより，物理レジスタ数によって生じる最適化間のトレードオフを，4 つ組中間コードレベルで緩和することが可能になる．

以下，2 章では我々が前提とする JAVA 言語[5)]用の Just-In-Time コンパイラの構成を述べる．3 章では DAG から 4 つ組コードを生成するアルゴリズムを述べる．4 章では，我々のアルゴリズムを IBM Developer's Kit for Windows，Java Technology Edition，Version 1.3 および IBM Developer's Kit for AIX，Java Technology Edition，Version 1.3 に実装し，Intel Pentium III プロセッサ[7)]および IBM POWER3 プロセッサ[8)]を対象とした性能評価を示す．5 章では，アーキテクチャ独立な最適化とレジスタプレッシャーのトレードオフを解消するためのコード生成アルゴリズムの拡張として，共通部分式削除の逆変換について述べる．6 章では物理レジスタ数制約を緩和するために DAG 上のコード変形や DAG からのコード生成を用いる既存の研究について述べる．7 章でまとめと今後の課題について述べる．

2. 最適化コンパイラの構成

本研究で我々が評価の対象とする DAG からの 4 つ組コード生成（以下，単に 4 つ組生成と呼ぶ）と，他の最適化パスとの順序を**図 1** に示す．図の Interme-

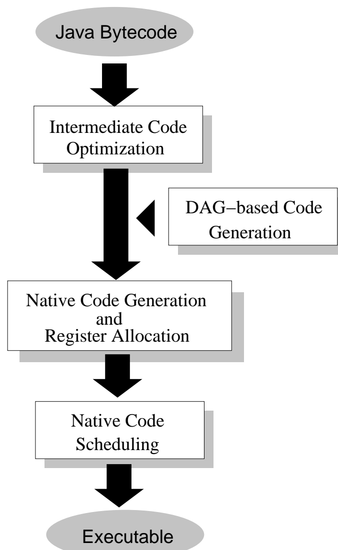

図 1 JAVA Just-In-Time コンパイラにおける最適化パスの構成
Fig. 1 Optimization passes in JAVA Just-In-Time compiler.

diate Code Optimization では，4 つ組中間コードで表現されたプログラムに対して，共通部分式の削除，コピー伝搬，定数の畳み込み，不要なコードの削除などアーキテクチャ独立な最適化が行われる．Native Code Generation and Register Allocation では，4 つ組中間コードからアーキテクチャごとの機械語命令を生成し，物理レジスタの割付を行う．Native Code Scheduling では固定長の命令ウィンドウ内で機械語命令を並べ替え，プロセッサの資源制約を考慮したコードスケジューリング最適化を行う．

1 章で述べたように，4 つ組中間コード上のアーキテクチャ独立な最適化とアーキテクチャ依存の最適化の間に，新たに中間コードを演算子の依存関係の DAG で表現し，DAG から再び 4 つ組中間コードを生成するパスを設ける．レジスタプレッシャーが高い部分では，レジスタ割付にとって有利な命令順に 4 つ組を生成する．すなわち，同時に生存する一時変数の数が少なくなるように 4 つ組生成を行う．4 つ組中間コード上で同時に生存する一時変数の数が物理レジスタ数より少なければ，後のレジスタ割付のパスではスピルコードを発生せずに物理レジスタを割り付けることができる．プログラム中のレジスタプレッシャーの低い部分では，コードスケジューラに有利になるような命令順で 4 つ組生成を行う．すなわち，一時変数の数を増やして命令レベル並列性を中間コード上で明示的に表現し，critical path 上の命令が連続になるように 4 つ組生成を行う．命令レベルの並列性が 4 つ組上で表現されていれば，レジスタ割付のパスで並列に実行できる命令に対して同じレジスタを割り付けることがない．critical path 上の命令が連続させせることにより，限られた命令ウィンドウ内でコードスケジューラが最適化を行う機会を増やす．

4 つ組生成は，他の最適化とは独立した 4 つ組中間コード上の変換であり，効率的な 1 パスのコードスケジューリングアルゴリズムを用いる．このようにパスを構成することによって，レジスタ割付においてレジスタプレッシャーを下げるためにコード移動を行ったり，コードスケジューリングにおいて並列性を抽出するためにレジスタをリネーミングしたりするような高いコストの解法を用いることなく，最適化間の物理レジスタ数制約に関するトレードオフを緩和することができる．また，4 つ組中間コード上の変換であるため，プロセッサの物理レジスタ数や 4 つ組の命令セットを変えることで，異なるアーキテクチャに容易に適用可能である．

今回我々が最適化の実装を行う IBM JAVA Just-In-Time コンパイラ[9),19)]は，JAVA 仮想マシンから起動され，図 1 のように，JAVA バイトコードを入力として JAVA 仮想マシンが動作しているプロセッサの機械語命令を生成する．コンパイル処理は JAVA バイトコードの実行時に行われるため，JAVA Just-In-Time コンパイラでは，最適化を含めたコンパイル処理自体が高速でなければならない．そのため，アーキテクチャ独立な最適化とアーキテクチャ依存最適化，あるいはレジスタ割付とコードスケジューリング最適化を完全に協調させて動かすことは，計算コストの観点から現実的でない．また，コードスケジューリング最適化はプロセッサの物理的な資源制約を解くので，対象となる命令列が長くなると大きな計算コストがかかるため，固定長のルックアヘッドウィンドウの範囲内で最適化が行われる．中間コードレベルでコード移動を行うことは，コードスケジューラの負担を軽減する．

3. DAG からの 4 つ組生成アルゴリズム

コード生成の手順は次の 3 段階に分かれる．

(1) 4 つ組表現の中間コードを DAG によって表現する．
(2) DAG の各頂点の演算子について，コード生成に必要なパラメータを計算する．
(3) レジスタプレッシャーを計算しながら，DAG から 4 つ組コード生成を行う．

以下で各段階の詳細を述べる．

3.1　DAGの生成

演算子の依存関係のDAG $G = (V, E)$ は，基本ブロック内の中間コードの依存関係を表現したグラフである．頂点 $v \in V$ は中間コードの演算子に対応する．辺 $e \in E \subset V \times V$ には以下の2種類がある．

データ依存　先行する頂点の演算子が生成する値を後継する頂点の演算子が使用することを表す．

順序制約　先行する頂点の副作用が後継する頂点の副作用より先に起きなければならないことを表す．

簡単のために，先行する頂点を持たないすべての演算子に先行する仮想的な頂点 $\top \in V$（トップ）と，後継する頂点を持たないすべての演算子に後継する仮想的な頂点 $\bot \in V$（ボトム）を定める．各頂点 v には演算子の結果が得られるまでのサイクル数 $cy(v)$ が割り当てられているとする．

図2 は4つ組表現の中間コードで記述されたプログラムの例である．このプログラムから，簡単のためt1–t8，x1–x4の演算だけをDAGで表現すると，依存関係のDAGは**図3**のように表される．JAVAプログラムでは配列アクセスにともなう例外の発生順序を守る必要がある．このプログラムには冗長な例外の検出を除去したり，プログラムの一部分を複製することにより例外の検出を除去するなどの最適化[9),19)]が適用可能であるが，本稿の主旨とは異なる最適化であるため，ここでは単にメモリアクセス間に順序制約を付加して例外の発生順序を保証する．

```
loop:
  t1 := a[i]
  t2 := a[i + 1]
  t3 := a[i + 2]
  t4 := a[i + 3]
  t5 := t1 * t2
  t6 := t1 * t3
  t7 := t2 * t4
  t8 := t3 * t4
  x1 := x1 + t5
  x2 := x2 + t6
  x3 := x3 + t7
  x4 := x4 + t8
  i := i + 1
  if (i < n) goto loop
```

図2　4つ組表現の中間コードで記述されたプログラムの例
Fig. 2　A sample program written in quadruple expression.

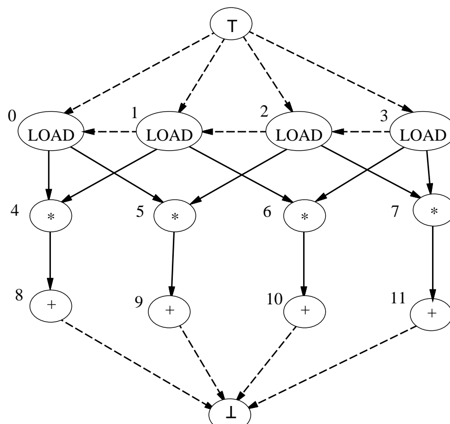

図3　例のプログラムのDAGによる中間コードの表現．実線矢印は演算子間のデータ依存，破線矢印は演算子間の順序制約を表す．
Fig. 3　DAG expression of the sample program. A solid arrow means data dependence between operators, and a dashed arrow means ordering constraint between operators.

3.2　パラメータの計算

3.1節で生成したDAGの各頂点に対して，以下の値を計算する．

トップからのレベル　各頂点 v のトップからのレベル $lt(v)$ を次のように求める．

$$lt(\top) = 0, \quad cy(\top) = 0,$$

$$lt(v) = \max_{p \in \mathrm{pred}(v)} (lt(p) + cy(p))$$

$\mathrm{pred}(v)$ は v に先行する頂点の集合を表す．

ボトムからのレベル　各頂点 v のボトムからのレベル $lb(v)$ を次のように求める．

$$lb(\bot) = 0,$$

$$lb(v) = \max_{s \in \mathrm{succ}(v)} (lb(s)) + cy(v)$$

$\mathrm{succ}(v)$ は v に後継する頂点の集合を表す．

自由度　$lt(\bot) = lb(\top)$ をDAGのcritical path長と呼び，$cp(G)$ で表す．各頂点の自由度（slackness）[11)] $sl(v)$ は，

$$sl(v) = cp(G) - lt(v) - lb(v)$$

自由度は4つ組生成を行うときの「余裕」を表している．トップからボトムに至るまでで，最もサイクル数が多くかかる経路（critical path）上の頂点の自由度は0である．頂点の自由度が大きいときは，その頂点の4つ組生成を遅らせても，生成されたコード列の実行サイクル数がcritical path長より長くなってしまう可能性が低いことを表している．

参照数の変化　頂点 v の参照数の変化 $\Delta ref(v)$ は，

$$\Delta ref(v) = \#(\mathrm{succ}_D(v)) - \#(\mathrm{pred}_D(v))$$

$\mathrm{succ}_D(v)$ および $\mathrm{pred}_D(v)$ は頂点 v に対してデータ依存で後継および先行する頂点の集合を表す．参照数が小さい頂点は，多くの先行する頂点の値を使い終わる可能性がある．参照数が大きい

表 1 トップからのレベル，ボトムからのレベル，自由度，参照数の変化の計算例
Table 1 level from the top, level from the bottom, slackness, and increase of reference count of the sample DAG.

vertex	0	1	2	3	4	5	6	7	8	9	10	11
$lt(v)$	3	2	1	0	4	4	3	2	5	5	4	3
$lb(v)$	3	4	5	6	2	2	2	2	1	1	1	1
$sl(v)$	0	0	0	0	0	0	1	2	0	0	1	2
$\Delta ref(v)$	+2	+2	+2	+2	−1	−1	−1	−1	−1	−1	−1	−1

頂点は，その頂点が生成する値が多くの頂点に参照される．したがって，長い間レジスタを占有する可能性がある．

図 3 の DAG で，演算にかかるサイクル数をすべて 1 とおくと，各頂点のパラメータは**表 1** のように計算される．

3.3 4 つ組生成

3.2 節で計算したパラメータを使って，トップから順に DAG の頂点を走査して 4 つ組生成を行う．4 つ組生成中に，現在生存中のレジスタの個数を計算し，4 つ組生成の方針を切り替える．基本的なアイディアは，レジスタプレッシャーが高いときには，まだ 4 つ組生成されていない頂点のうち，レジスタを多く解放する頂点から先に 4 つ組生成し，レジスタプレッシャーが低いときには，DAG の critical path 上にある頂点が連続するように 4 つ組生成を行うというものである．

以下のデータ構造は，現在の 4 つ組生成の状態を表す．

- $u(G)$：まだ 4 つ組生成されていない頂点の集合
- $r(G)$：次に 4 つ組生成可能な頂点の列
- $a(G)$：すでに 4 つ組生成された頂点の列
- reg：現在生存中のレジスタの数

4 つ組生成は，$r(G)$ に属する頂点を 1 つずつ選び，$a(G)$ に追加する手続きを繰り返すことで進行する．$u(G)$ と $r(G)$ と $a(G)$ は disjoint である．以下のデータ構造は頂点 v の状態を表す．

- $ap(v)$：頂点 $v \in u(G)$ に先行する頂点で，4 つ組生成が終了していない頂点の数
- $au(v)$：頂点 $v \in a(G)$ にデータ依存で後継する頂点で，4 つ組生成が終了していないものの数
- $\Delta reg(v)$：頂点 $v \in r(G)$ が新たに増やすデータ依存の数

$v \in u(G)$ のうち $ap(v) = 0$ である頂点 v は 4 つ組生成が可能であるため $u(G)$ から取り除かれ $r(G)$ に入る．$au(v)$ はすでに生成された 4 つ組が生成した値によるレジスタプレッシャーを表す値である．$\Delta reg(v)$ は v を 4 つ組生成したときに新たに必要になるレジスタの個数である．

4 つ組生成のアルゴリズムは以下のように表される．

(1) 上記のデータ構造を初期化する．
$a(G) = \{\}$
$r(G) = \{\top\}$
$u(G) = V - \{\top\}$
$ap(v) = \#\{e \in E \mid e = (p, v), p \in V\}$
$au(v) = \#\{s \in \mathrm{succ}_D(v)\}$
$\Delta reg(\top) = 0$
reg は基本ブロックの外で定義され，基本ブロックの中で使用されている変数の数に初期化する．

(2) $r(G)$ の先頭の頂点 v を取り出し，$a(G)$ の最後に加える．

(3) 頂点 v に後継する辺 $\{e \in E | e = (v, s), s \in u(G)\}$ について，$ap(s) := ap(s) - 1$．

(4) 3 で $ap(w) = 0$ となる頂点 w があれば，$u(G)$ から取り除き $r(G)$ に追加する．

(5) 頂点 v に先行する頂点 $p \in \mathrm{pred}_D(v)$ について，$au(p) := au(p) - 1$

(6) $reg := reg + \Delta reg(v)$．

(7) $v' \in r(G)$ について，使用中のレジスタの個数の増分 $\Delta reg(v')$ を計算する．
$\Delta reg(v') = nval(v') -$
$\#\{p \in \mathrm{pred}_D(v') \mid au(p) = 1\}$
$nval(v')$ は頂点 v' が値を生成するために確保するレジスタの数である．

(8) $r(G)$ が空であれば終了．

(9) reg がプロセッサの物理レジスタ数より小さい場合は，並列性の抽出を優先して $r(G)$ を整列する．優先順位を決めるために，以下の 2 つのパラメータを使用する．
SLK $sl(v')$ が小さい方の頂点を優先する．
BTM $lb(v') + sl(v')/2$ の値が大きい方の頂点を優先する．ボトムからのレベルに自由度を加えたのは，負荷が時間方向に均等に分布するようにするためである．

この場合は，使用可能なレジスタが十分あるので，critical path 上にある頂点を優先し，コードスケジューリング最適化で演算器の遅延を隠

蔽できるように，4 つ組生成を行う．

(10) reg がプロセッサの物理レジスタ数以上である場合は，使用レジスタ数の最小化を優先して $r(G)$ を整列する．優先順位を決めるために，以下の 2 つのパラメータを使用する．

REG $\Delta reg(v')$ が小さい方の頂点を優先する．

REF $\Delta ref(v')$ が小さい方の頂点を優先する．

使用レジスタ数の観点から順番が決まらない場合は，並列性の抽出を優先するときと同じ基準を用いる．この場合は，使用可能なレジスタ数が少ないので，なるべくレジスタを多く解放し，新しく確保しないような頂点を優先して 4 つ組生成する．本手法では $r(G)$ の中から，次に生成すべき頂点によるレジスタ数の増減を，1 つだけ先読みしている．これを複数の頂点の列について拡張する方法も考えられるが，検討すべき組合せが非常に多くなるので，コンパイル時間と最適化の効果の兼ね合いから，1 つ先読みする方法が有効である．使用可能なレジスタ数が少なくなったことを早めに検出するために，実際の物理レジスタ数より少ない閾値を使用する．

(11) (2) へ戻る．

図 3 の DAG に対して，レジスタ数が 8 のプロセッサとレジスタ数が 32 のプロセッサで 4 つ組の生成を行ったとする．4 個の変数がイテレーションを超えるデータ依存を持っているので，初期状態では $reg = 4$ である．4 つ組生成方針を切り替える閾値を (物理レジスタ数) − 2 に設定すると，レジスタ数が 8 のプロセッサでは，4 つ組生成が終了したときには，

$$a(G) = \{\top, 3, 2, 7, 11, 1, 6, 10, 0, 5, 4, 9, 8, \bot\}$$

の順になる．レジスタ数が 32 のプロセッサでは，

$$a(G) = \{\top, 3, 2, 1, 0, 5, 4, 6, 7, 9, 8, 10, 11, \bot\}$$

である．後者は 5 個の一時変数を必要とするが，前者では 3 個で済んでいる．

4. 性能評価

IBM Developer's Kit for Windows, Java Technology Edition, Version 1.3 および IBM Developer's Kit for AIX, Java Technology Edition, Version 1.3 に前述のアルゴリズムを実装し，2 つの異なるプラットフォーム，Intel Pentium III プロセッサおよび IBM POWER3 プロセッサを対象として性能評価を行った．ベンチマークプログラムとして，JAVA 言語によるベンチマークセットの 1 つである jBYTEmark[3] および，SPEC JVM98 ベンチマーク[18] の中から mpegaudio ベンチマークを選んだ．jBYTEmark は整列，ビットフィールド演算，符号処理，暗号処理など整数および浮動小数点数の計算を行う 10 個のベンチマークの組である．mpegaudio ベンチマークは MPEG Layer-3 形式の音声データのデコードを行う．これらのベンチマークはいずれもカーネルに整数および浮動小数点数演算のループを含むため，基本ブロック内の 4 つ組生成やコードスケジューリング最適化の効果が観測できる．以下の表中で，mpegaudio ベンチマークの性能は実行時間（小さい方が性能が高い），jBYTEmark の性能は整数演算および浮動小数点数演算ごとのインデクス（大きい方が性能が高い）で示す．

POWER3 プロセッサは 32 個の整数レジスタと 32 個の浮動小数点数レジスタを持つ．このレジスタ数は基本ブロック内の命令レベル並列性を抽出するには十分多いので，DAG からの 4 つ組生成では並列性の抽出が優先される．POWER3 はスーパスカラプロセッサで，命令の発行は in-order，命令の実行は out-of-order である．命令の発行が in-order であるため，コードスケジューリング最適化の効果が顕著である．我々は，並列性抽出を優先した場合のパラメータの影響を調べるため，以下の組合せについて実験を行った．

BASE DAG からの 4 つ組生成を行わないもの．

SLK-BTM 3 章のアルゴリズムで，並列性の抽出時に SLK を BTM より優先するもの．

BTM-SLK 3 章のアルゴリズムで，並列性の抽出時に BTM を SLK より優先するもの．

MIN つねに使用レジスタ数を最小化する 4 つ組生成を行うもの．

NSCH BASE に加えて，レジスタ割付の後で行われる機械語レベルのコードスケジューリング最適化を行わないもの（比較のため）．

表 2 に 400 MHz の POWER3 プロセッサでベンチマークを行ったときの結果を示す．4 つ組生成方針を切り替える閾値は (物理レジスタ数) − 2 に設定した．各行は性能を測定したときのパラメータを示す．各列は左から mpegaudio の実行時間，jBYTEmark の整数演算，浮動小数点数演算のインデクス，mpegaudio，jBYTEmark の BASE からの性能向上の割合を示している．自由度を優先した 4 つ組生成（SLK-BTM）はあまり良い結果を与えていない．一方ボトムからのレベルと自由度の和を優先したもの（BTM-SLK）は 3〜6%の性能向上を得た．この現象の原因は，中間コードレベルの DAG の critical path の見積りが不正確である場合，自由度だけを優先すると，自由度が少しでも大きい経路の頂点はすべて critical path の

表 2 4 つ組生成の方針を変化させたときの，POWER3 プロセッサ（400 MHz）における jBYTEmark のインデクスおよび mpegaudio ベンチマーク（test mode, size = 100）の実行時間（秒）

Table 2 Indices of jBYTEmark and best execution time of mpegaudio benchmark (test mode, size = 100, in seconds) on POWER3 processor (400 MHz), when we change quadruple generation policy.

	mpegaudio	jBYTEmark		performance improvement		
	best run	int index	fp index	mpeg	jb-int	jb-fp
BASE	9.880	127.10	83.03	–	–	–
SLK-BTM	10.689	128.97	79.30	−8.18%	1.47%	−4.50%
BTM-SLK	9.365	131.27	88.20	5.22%	3.28%	6.22%
MIN	9.444	130.93	82.87	4.41%	3.02%	−0.20%
NSCH	10.766	124.03	78.23	−8.97%	−2.41%	−5.78%

表 3 4 つ組生成の方針を変化させたときの，Pentium III プロセッサ（600 MHz）における jBYTEmark のインデクスおよび mpegaudio ベンチマーク（test mode, size = 100）の実行時間（秒）

Table 3 Indices of jBYTEmark and best execution time of mpegaudio benchmark (test mode, size = 100, in seconds) on Pentium III processor (600 MHz), when we change quadruple generation policy.

	mpegaudio	jBYTEmark		performance improvement		
	best run	int index	fp index	mpeg	jb-int	jb-fp
BASE	11.456	145.33	91.90	–	–	–
REG-REF	11.447	144.00	92.77	0.08%	−0.92%	0.94%
REG	12.649	141.89	91.31	1.02%	−2.37%	−0.64%
MAX	11.914	138.40	91.77	−3.99%	−4.77%	−0.15%
NSCH	11.422	141.33	91.33	0.30%	−2.75%	−0.62%

頂点より後で 4 つ組生成されてしまうことである．すなわちボトムからのレベルを優先した 4 つ組生成の方が，実行時間の見積りの誤りに関しては安定している．つねに使用レジスタ数を最小化する方針（MIN）で 4 つ組生成を行う場合でも，2〜5%の性能向上がみられる．これは，使用レジスタ数の増加に関する優先度が同じ演算子に対しては，命令レベル並列性で優先度が付けられるためである．POWER3 プロセッサは命令のリオーダーバッファのサイズが小さく，演算ユニットを効率良く使用するには，コードスケジューラが，並列性のある命令列ができるだけ連続して発行されるように命令を並べ替えなければならない．リオーダーバッファが小さいため，機械語レベルのコードスケジューリング最適化の効果は最大 10%と非常に大きい（NSCH）．レジスタ割付でレジスタの逆依存や出力依存などの不必要な順序制約が導入されると，スケジューリングによる命令並べ替えが妨げられる．したがって，DAG からの 4 つ組生成で命令レベル並列性を抽出することによって，レジスタ割付とコードスケジューリングの間の干渉を緩和し，コードスケジューリングの効果を大きく上げることができる．

Pentium III プロセッサは 8 個の整数レジスタと 8 個の浮動小数点数レジスタを持つ．このレジスタ数は基本ブロック内の命令レベル並列性に対してはかなり少ないので，DAG からの 4 つ組生成ではできるだけ使用レジスタ数を少なくしてスピルコードを最小化する必要がある．我々は，レジスタプレッシャーの緩和を優先した場合のパラメータの影響を調べるため，以下の組合せについて実験を行った．

BASE DAG からの 4 つ組生成を行わないもの．

REG-REF 3 章のアルゴリズムで，使用レジスタ数の最小化時に REG を REF に優先するもの．

REG 3 章のアルゴリズムで，使用レジスタ数の最小化時に REG だけを用いるもの．

MAX つねに並列性抽出を優先する 4 つ組生成を行うもの．並列性の抽出時には BTM を SLK に優先する．

NSCH BASE に加えて，レジスタ割付の後で行われる機械語レベルのコードスケジューリング最適化を行わないもの（比較のため）．

表 3 に 600 MHz の Pentium III プロセッサでベンチマークを行ったときの結果を示す．4 つ組生成方針を切り替える閾値は (物理レジスタ数) − 2 に設定した．各行は性能を測定したときのパラメータを示す．各列の意味は表 2 と同様である．3 章のアルゴリズム（REG-REF）によって目立った性能向上を得ることはできなかった．参照数を用いることの効果（REG）もこの数字からは判断できない．しかし，つねに並列性

抽出を優先した場合（MAX）は 4〜5%の性能低下が生じることから，与えられた DAG からできるだけレジスタプレッシャーの低い 4 つ組コード列を生成するという目標は達成されているといえる．Pentium III も POWER3 と同様スーパスカラプロセッサであるが，POWER3 に比べて非常に大きなリオーダーバッファを持つ．このことは，ハードウェアによってコードスケジューリング最適化を実行時に行っていることに相当する．したがって，コードスケジューリング最適化による効果自体も POWER3 の場合ほど顕著ではない（NSCH）．

POWER3 でははっきりした効果が現れたが，Pentium III でほとんど性能向上がみられなかった原因の 1 つとして考えられるのは，DAG からの 4 つ組生成を行わない，元々の 4 つ組中間コードの順序のレジスタプレッシャーが低いという可能性である．JAVA Just-In-Time コンパイラは，JAVA バイトコードを 4 つ組中間コードに変換して最適化を行う．JAVA バイトコードはスタック上で演算を行うので，中間コードの順序を途中で大きく入れ替えない限り，スタックに対応した一時変数の生存区間は短い．このような中間コードには，比較的少ない数でのレジスタ割付が可能であり，また明示的に中間コードに表現されている命令レベル並列性は低い．このことは，並列性の抽出が POWER3 プロセッサに対して有効であることに関連すると考えられる．

5. 4 つ組生成アルゴリズムの拡張

アーキテクチャ独立な最適化の中でも，共通部分式削除は変数の参照数を増やし，一時変数の生存区間を長くする．本章では，3 章で示した 4 つ組生成アルゴリズムを拡張して，データ依存の干渉度が高くなってレジスタが不足したときに，共通部分式削除の逆変換を行い干渉度を下げる手法を述べる．

3.3 節のアルゴリズムの 10 で，以下のように逆変換を行う．

（1） $v \in a(G)$ のうち，以下の条件が成り立つ頂点の集合 $C \subset V$ を求める．

$au(v) > 0,$

$\forall p \in \mathrm{pred}_D(v), au(p) > 0,$

$\#\{s \in (a(G) \cap \mathrm{succ}_D(v))\} \geq 1$

（2） C の中で自由度が最も大きい頂点 c を選ぶ．

（3） c を複製した頂点 c' を生成し，$(u(G) \cup r(G)) \cap \mathrm{succ}_D(c)$ に属する頂点 s に対して，以下のように辺を張り替える．

$e = (c, s) \rightarrow (c', s)$

（4） c' を $u(G)$ に加える．$au(c) := 0$ となる．

6. 関連研究

DAG からのコードスケジューリングによって使用レジスタ数を最小化する問題は NP 完全である．最適解を求めるアルゴリズムとして，動的計画法を用いた $O(n2^{2n})$ の手法が提案されている[10]．彼らの手法では約 50 命令までの問題の最適解を求めることができる．コンパイル時間の爆発を防ぐためには，計算量およびメモリ量は入力となる命令列の長さの線形オーダーに近いことが望ましい．そのため，実際のシステムではヒューリスティクスアルゴリズムが用いられる．

ヒューリスティクスを用いて DAG 上のコード変形と DAG からの 4 つ組コード生成でレジスタプレッシャーの緩和を行うアプローチでは，DSP のアキュミュレータの演算とスピルコードを最小化する研究[13]がある．彼らが用いるヒューリスティクスは，1 パスのコードスケジューリングアルゴリズムで，対象プロセッサのモデルは異なるが，使用レジスタ数を最小化するための基本的な方針は我々と同じである．彼らの研究では，乗算を含まない線型変換に問題を限定することで，最適なコード列を求める．同様に DSP を対象としたものとして，分枝限定法を用いて DSP のアキュミュレータのスピルを最小化するアルゴリズム[12]が提案されている．我々の目標は，プロセッサや入力に特に制限を設けず，一般の条件でレジスタ割付やコードスケジューリングなどのアーキテクチャ依存の最適化を支援することである．

Chen らによる研究[4]では，レジスタプレッシャーを考慮した 1 パスのプリパススケジューリングの代わりに，トップダウンのリストスケジューリングとレジスタプレッシャーを考慮したボトムアップのコード再構成の 2 パスを行うことを提案している．コードの再構成では動的にスケジューリング方針を切り替えるのではなく，レジスタプレッシャーを計算しながらコード移動を行う．一般的に，コード移動によってレジスタプレッシャーを制御する機能は，生成コードを改善するうえで非常に重要である．性能評価で見たように，コードスケジューリングとレジスタ数制約のいずれが支配的であるかはアーキテクチャによって異なる．したがって，最適なパスの構成はアーキテクチャにより異なる可能性があり，どのようなパスを構成するかは検討の余地がある．

7. まとめと今後の課題

本稿では，アーキテクチャ独立な最適化とアーキテ

クチャ依存最適化の間に，DAG による表現から 4 つ組生成を行うパスを設けることで，4 つ組中間コード上の最適化，レジスタ割付，コードスケジューリングの間の物理レジスタ数の制約によるトレードオフを効率的に緩和する手法を示した．我々は本手法を IBM Developer's Kit for Windows，Java Technology Edition，Version 1.3 および IBM Developer's Kit for AIX，Java Technology Edition，Version 1.3 に実装し，POWER3 プロセッサと Pentium III プロセッサを対象としてベンチマークによる性能評価を行った．POWER3 プロセッサでは，ボトムからのレベルを用いた並列性の抽出により，約 5%の性能向上を得た．Pentium III プロセッサでは，現状からの目立った性能向上は見られなかったものの，使用レジスタ数を最小化するアルゴリズムが機能していることを確認した．さらに，アルゴリズムを拡張することによって，共通部分式削除などの中間コード最適化の逆変換を組み込むことができることを示した．

本手法によって，レジスタプレッシャーが高い基本ブロックに対して，スピルコードをなるべく少なくするよう 4 つ組生成することが可能である．しかし，x86 アーキテクチャのように，物理レジスタの数自体が少ない場合は，最適なコードを生成するためには一時レジスタの数を非常に正確に見積もる必要がある．共通部分式削除など，1 パスの 4 つ組生成時に逆変換を行える最適化は，本手法によって統一的に扱うことができるが，配列アクセスのストライディングなど，ループ構造に関連するものは，4 つ組生成時に簡単にキャンセルすることができない．今後，より広いクラスの最適化に適用できるように，改良を進める予定である．

謝辞 本研究を行うに際して数々の示唆と助言をいただいた日本 IBM 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はじめ同グループの研究員の方々に感謝します．

参 考 文 献

1) Aho, A.V., Johnson, S.C. and Ullman, J.D.: Code Generation for Expressions with Common Subexpressions, *J. ACM*, Vol.24, No.1, pp.146–160 (1977).
2) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Co., Reading, MA (1986).
3) BYTE magazine online: BYTE Benchmarks (1998). available at http://www.byte.com/bmark/bmark.htm.
4) Chen, G. and Smith, M.D.: Reorganizing Global Schedules for Register Allocation, *Proc. 1999 International Conference on Supercomputing*, pp.408–416 (1999).
5) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley, Reading, MA (1996).
6) Gupta, R. and Bodik, R.: Register Pressure Sensitive Redundancy Elimination, *Proc. 8th International Conference on Compiler Construction*, LNCS, Vol.1575, pp.107–121 (1999).
7) Intel Corporation: Pentium Faimily User's Manual (1994).
8) International Business Machines Corporation: RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide (1998).
9) Ishizaki, K., Kawahito, M., Yasue, T., Takeuchi, M., Ogasawara, T., Suganuma, T., Onodera, T., Komatsu, H. and Nakatani, T.: Design, Implementation, and Evaluation of Optimizations in a Just-in-Time Compiler, *Proc. ACM SIGPLAN Java Grande Conference*, pp.119–128 (1999).
10) Keßler, C.W.: Scheduling Expression DAGs for Minimal Register Need, *Computer Languages*, Vol.24, No.1, pp.33–53 (1998).
11) 小松秀昭，神力哲夫，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レジスタ割り付け手法，情報処理学会論文誌，Vol.36, No.12, pp.2819–2829 (1995).
12) Liao, S., Devadas, S., Keutzer, K., Tjiang, S. and Wang, A.: Code Optimization Techniques for Embedded DSP Microprocessors, *Proc. 32nd Design Automation Conference*, pp.599–604 (1995).
13) Mehendale, M., Venkatesh, G. and Sherlekar, S.D.: Optimized Code Generation of Multiplication-free Linear Transforms, *Proc. 33rd Design Automation Conference*, pp.41–46 (1996).
14) Morel, E. and Renvoise, C.: Global Optimization by Supression of Partial Redundancies, *Comm. ACM*, Vol.22, No.2, pp.96–103 (1979).
15) Norris, C. and Pollock, L.L.: An Experimental Study of Several Cooperative Register Allocation and Instruction Scheduling Strategies, *Proc. 28th Annual International Symposium on Microarchitecture*, pp.169–179 (1995).
16) Pinter, S.S.: Register Allocation with Instruction Scheduling: a New Approach, *Proc. Conference on Programming Language Design and Implementation*, pp.248–257 (1993).

17) Redziejowski, R.R.: On Arithmetic Expressions and Trees, *Comm. ACM*, Vol.12, No.2, pp.81–84 (1969).
18) Standard Performance Evaluation Corporation (SPEC): SPECjvm98 Benchmarks (1998). http://www.spec.org/osg/jvm98.
19) 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.175–193 (2000).

(平成 12 年 5 月 26 日受付)
(平成 12 年 9 月 8 日採録)

稲垣　達氏

1970 年生．1995 年東京大学大学院理学系研究科情報科学専攻修士課程修了．1998 年東京大学大学院理学系研究科情報科学専攻博士課程退学．同年日本 IBM 入社．現在日本 IBM 東京基礎研究所先任研究員．JAVA Just-In-Time コンパイラの開発に従事．

古関　　聰（正会員）

1969 年生．1994 年早稲田大学大学院理工学研究科電気工学専攻修士課程修了．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 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．

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）．