部分冗長除去に基づく大域命令スケジューリング

勝原達也[†] 滝本宗宏[††]

命令レベル並列性を高める有効な手法の 1 つとして，命令スケジューリングがある．特に，投機的実行を許す命令スケジューリングは，さらに並列度を高められる点で効果的であることが知られている．投機的実行は，元のプログラムに存在しなかった冗長性を導入する可能性があるので，1 つの命令をスケジュールするたびに共通部分式の削除を適用することが効果的である．本発表では，共通部分式の削除で取り除くことができない部分冗長性を扱う部分冗長除去に注目し，部分冗長除去に基づく大域命令スケジューリングを提案する．本手法は，あるプログラム点に対して，投機的命令スケジューリングを含むコードの上方移動を行うごとに，部分冗長除去のデータフロー解析を適用する．解析結果から最適化効果が認められる場合，実際にプログラムを変更することで，命令スケジューリングを実現する．部分冗長除去の性質は，プログラムの実行パスを長くしないことを保証し，大域命令スケジューリングで問題となる補償コードを最適なプログラム点に自動的に挿入する．また，本手法は，ループ構造を認識することなく，ループスケジューリング手法の 1 つである，ループシフティングを実現することができる．さらに本論文では，本手法を COINS コンパイラ・インフラストラクチャ上で実装し，最適化効果を検証する．

Global Instruction Scheduling Based on Partial Redundancy Elimination

Tatsuya Katsuhara[††] and Munehiro Takimoto[††]

Instruction scheduling is one of the effective techniques to increase Instruction-level parallelism. Especially, the instruction scheduling which allows speculative execution is known as a technique which exposes more parallelism in a program. Since the speculative execution can introduce new redundant expressions as a second order effect, eliminating the redundant expressions using common sub-expression elimination (CSE) opens a way for scheduling other operations. We propose an instruction scheduling approach based on partial redundancy elimination (PRE) which eliminates more redundant expressions than CSE does. PRE can eliminate not only totally redundant expressions which CSE can eliminate but also partially redundant expressions, therefore leads to more opportunities to schedule. In addition to that, we focus on a property such that PRE eliminates redundant expressions based on code motion. This property enables determining whether scheduling to a specific idol resource is effective or not, and automatically inserting any compensation code to optimal points after scheduling. Hence, our scheduling approach guarantees that any execution paths are not lengthened. Our approach can also be applied to any program structures, and achieve a loop scheduling, i.e. loop shifting, without identifying loop structure.

1. はじめに

VLIW プロセッサやスーパスカラプロセッサのような，複数命令を同時実行できるマイクロアーキテクチャの性能向上では，**命令レベル並列性**（Instruction-Level Parallelism）を高めることが重要である．コンパイラのコード最適化において，命令レベル並列性を高めようとする場合，命令を並列に実行できるように並べ替える，**命令スケジューリング**（instruction scheduling）を用いるのが一般的である．

命令スケジューリングは，分岐を含まない基本ブロックを対象とした**局所命令スケジューリング**が中心だが，解析範囲が狭いので，並列性の向上には限界がある．一方，解析対象がプログラム全体である**大域命令スケジューリング**は，基本ブロックを超えたスケジューリングを行うことができるので，局所命令スケジューリングに比べ，広い範囲から並列性を抽出できるが，プログラムの意味を保存するために**補償コード**（compensation code）の挿入や，ループ構造の認識といった，付加的な解析が必要になる．

大域命令スケジューリングの中でも，投機的な実行

† 株式会社野村総合研究所
Nomura Research Institute, Ltd.
†† 東京理科大学理工学部情報科学科
Department of Information Sciences, Faculty of Science and Technology, Tokyo University of Science

を許して命令をスケジュールする，**投機的命令スケジューリング**（speculation）[18] は，さらに命令レベル並列性を向上させる手法として知られている．投機的命令スケジューリングは，いくつかの実行パス上に，新たに命令を移動させるので，冗長な式を増加させる可能性がある．S. Gupta らは，投機的命令スケジューリングを行うたびに，新たに生じた冗長性を**共通部分式の削除**（Common Sub-expression Elimination，以降 CSE）[18] で除去する，**動的 CSE**（dynamic cse）[16] を提案した．動的 CSE は，命令スケジューリングと冗長除去を組み合わせた，初めての手法である．

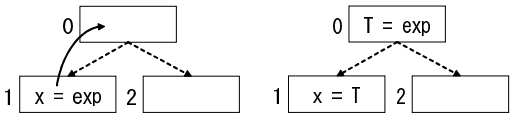
しかしながら，プログラムに存在する冗長性の中で，いくつかの実行パスでは冗長であるが，他の実行パスでは冗長でないという，**部分冗長性**（partial redundancy）は，CSE で扱うことができない．この部分冗長性は，**部分冗長除去**（Partial Redundancy Elimination，以降 PRE）[4,9,21] によって除去できる可能性がある．PRE は，冗長性がない実行パス上に命令を挿入することで，冗長な操作を除去することができる．このような変形は命令を制御フローと逆向きに移動することと同じであり，**コード巻上げ**（code hoisting）と呼ばれる．

本研究では，投機的命令スケジューリングの結果生じる冗長性を，CSE よりも PRE の方が多く除去できる点と，PRE が行うプログラム変換が，命令スケジューリングで行う命令の移動に類似している点に着目し，PRE に基づく大域命令スケジューリングを提案する．

本手法で，PRE は，単に投機的命令スケジューリングで生じた冗長性を除去するために働くのではなく，命令スケジューリングの一部として機能する．これは，命令スケジューリングに PRE の性質を組み込むことを意味し，次の利点を生じる．

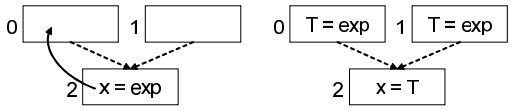
(1) 最適化効果のないスケジューリングを行わない．
(2) 大域命令スケジューリングで問題となる補償コードの挿入を自動的に行う．
(3) ループ構造を認識せずに，ループスケジューリングの 1 つである，**ループシフティング**（loop shifting）[17] を実現する．

本論文の以降の構成は，次のとおりである．2 章では，投機的命令スケジューリングについて述べる．3 章で，共通部分式の削除を説明したのち，投機的スケジューリングが導入する冗長性と，それを除去する動的 CSE の効果について述べる．次に 4 章で簡単に PRE の説明を行い，5 章で本手法のアルゴリズムの説明を行う．6 章ではループシフティングの説明と，本手法の枠組みでどのように実現されるかを述べる．7 章の実装，8 章の評価において，本手法の実現性と効果について示したのち，9 章で今後の課題ついて考察し，最後に関連研究とまとめを述べる．



(a) 移動前 (b) 安全な移動

図 1 投機的コード移動
Fig. 1 Speculative code motion.



(a) 移動前 (b) 安全な移動

図 2 合流点を越える投機的コード移動
Fig. 2 Speculative code motion across join node.

2. 投機的命令スケジューリング

投機的命令スケジューリングは投機的コード移動の 1 種であり，**投機的実行**（speculative execution）[1,18] を許す命令スケジューリングである．投機的実行とは，ある実行パス上の命令が実行されると仮定して，本来のプログラム点よりも先行して命令を実行することである．実行を仮定した実行パス上の命令とは別の実行パス上の命令が実行された場合，本来は実行されないはずの命令を実行パスに導入することになる．

図 1 に投機的コード移動の例を示す．分岐も合流もない領域を節として，それらの間を制御フローを表す有効辺で結んだグラフは**制御フローグラフ**（Control Flow Graph，以降 CFG）と呼ばれる．以降，特に断らない限り，CFG 節は基本ブロックとする．図 1 の CFG で，矩形は節を，点線矢印は制御フローを表す．図 1 (a) の節 1 の命令を節 0 へ移動させることを考える．移動前は分岐の結果に命令の実行が依存していたが，図 1 (b) のように，移動後は分岐の結果にかかわらず実行される，この移動は投機的コード移動である．この例では，節 2 に到達する変数 `x` の値を書き換えないように，一時変数 `T` を導入し，元の式 `exp` を `T` で置き換えている．もし節 0，2 のパスにおいて，変数 `x` が生存していないなら，一時変数 `T` と節 1 のコピー代入を導入せずに，投機的コード移動を行うことができる．この投機的コード移動は，制御の分岐点を越えるような後向き（backward）移動に分類される．

次に，**図 2** (a) で節 2 の命令を節 0 へ移動させるこ

とを考える．今まで命令があったプログラム点より先行して命令を実行することになるので，この移動は投機的な性質を持つ．しかし，命令を移動させると，節1，2のパスにおいて，命令が実行されなくなり，プログラムの意味が変わるので，図2(b)に示すように，節1に補償コードを挿入する必要がある．このような，合流点を越える投機的コード移動は，**条件節投機実行**（conditional speculation）[18]と呼ばれる．

投機的コード移動は，投機的に命令が移動されたパス上では，命令レイテンシを隠蔽する点で有益である．しかし別のパスにとっては必要のない計算が増える可能性があり，性能低下につながる可能性がある．つまり，投機的に移動する対象の命令はレイテンシが大きいほど有益であり，命令を移動したパスの実行頻度が高いほど，別のパスに挿入された補償コードの影響を少なくする．

投機的コード移動を命令スケジューリングに組み込んだ投機的命令スケジューリングでは，計算資源が空いている**空きリソース**（idle resource）に命令をスケジュールして並列化が可能である．つまり，図1(b)においては，実行パス0，1と実行パス0，2で性能は低下しない．また，図2(b)においては，実行パス0，2で性能は低下しないが，実行パス1，2では移動する式がもともと並列化されていた場合，補償コードが空きリソースに割り当てられなければ性能が低下する可能性がある☆．

図3に投機的命令スケジューリングの効果を示す．演算子ラベルのついた円と点線で囲まれた円は，それぞれ，命令と空きリソースを表す．実線矢印はデータ依存を表す．図3(a)の節0で命令＋の資源が空いているので，節2の命令＋を節0にスケジュールできる．このようなスケジューリングは，実行パス0，1，3に新しく命令＋を導入するので，投機的命令スケジューリングである．節1の命令－も同様に節0にスケジュールしたとすると，図3(b)のように変形することができる．この投機的命令スケジューリングの結果，節1に命令－の空きリソースが新たに生じるので，節3の命令－は，図3(c)のように，節1，2にスケジュールできる．

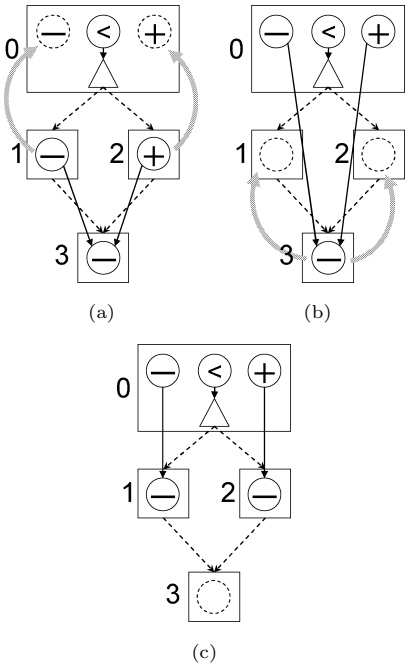


図3 投機的命令スケジューリング
Fig. 3 Speculation.

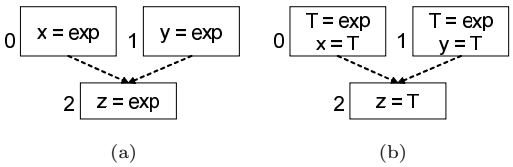


図4 共通部分式の削除
Fig. 4 Common sub-expression elimination.

3. 動的 CSE

3.1 共通部分式の削除

あるプログラム点 p に式 e が存在し，プログラムの開始点からプログラム点 p へと至る実行パス上に同じ式 e が出現するとき，式 e が最後に出現するプログラム点 p' とプログラム点 p との間で，式 e のオペランドの値が変更されない場合，式 e は，その実行パス上で冗長である，あるいはプログラム点 p の式 e は，プログラム点 p' の式 e に対して冗長であるという．

プログラム開始点から，あるプログラム点の式に到達するすべての実行パス上で式が冗長ならば，それを**共通部分式**（common sub-expression），または**全冗長**（totally redundant）な式と呼ぶ．CSEは，先行するプログラム点で計算済みの式の値を一時変数に保持しておき，後続の共通部分式を一時変数で置き換えることによって，共通部分式を除去する．

図4にCSEの簡単な例を示す．図4(a)の節2における式 `exp` は，節0，1から節2へ至るすべての実行パス上でオペランドが変更されないので，節2の式

☆ 投機的命令スケジューリングの結果生じるコピー代入の命令レイテンシが，十分小さいと仮定する．

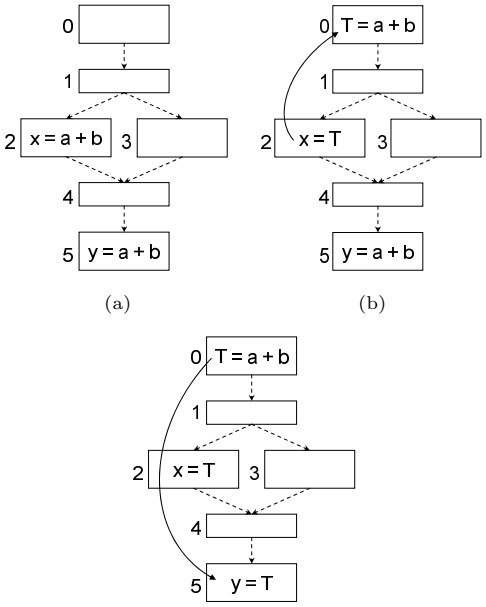


図 5 動的 CSE
Fig. 5 Dynamic CSE.

exp は全冗長である．全冗長な式は CSE を適用することで，図 4 (b) のように除去することができる．CSE を行うと，図 4 (b) の節 2 の z=T のようなコピー代入を導入することになるが，コピー代入は，変数 z のオペランドあるいは実引数としての出現を T で置き換えていく**コピー伝播**（copy propagation）[2),18)] や，**レジスタ彩色手法**（register coloring）で用いられる，**変数合体**（coalescing）[18)] によって取り除かれることが期待できる．

3.2 動的 CSE

動的 CSE は，投機的命令スケジューリングによって生じる冗長な式を，CSE を適用して除去するコード最適化である．各命令をスケジュールするたびに，CSE を行うことから，動的 CSE と呼ばれる．

動的 CSE の効果を**図 5** に示す．図 5 (a) の CFG において，節 2 の式 a+b を節 0 に投機的にスケジュール☆すると，図 5 (b) のように，節 5 の式 a+b が節 0 に対して冗長になる．すなわち，スケジューリングは，冗長な式を増す可能性がある．新たに生じた冗長な式は，CSE を適用して図 5 (c) の節 5 のように除去することができる．

このようなスケジューリングと冗長除去を組み合わせた手法は，実行パスをさらに短くすることに貢献する．

☆ 冗長除去の対象は式であり，命令スケジューリングの対象は命令である．本論文では，冗長性を考慮して命令をスケジュールすることを，「式をスケジュールする」と表現する．

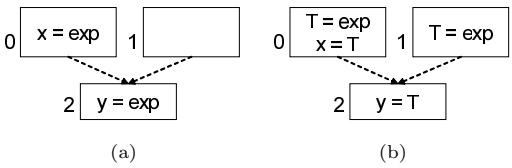


図 6 部分冗長除去
Fig. 6 Partial redundancy elimination.

4. 部分冗長除去

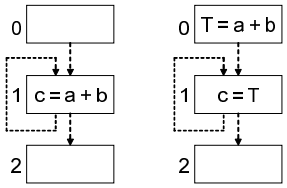
プログラムの開始点から，ある特定の式に到達するすべての実行パス上に同一の式が出現する場合，その式は全冗長な式であり，CSE を適用することによって除去することができる．これに対して，ある特定の式の出現が，その式を含むいくつかの実行パスでは冗長だが，他の実行パスでは冗長でない場合，その式は**部分冗長**（partially redundant）な式であるといい，CSE で除去することができない．PRE は，全冗長な式に加えて，部分冗長な式も除去することができる．

図 6 (a) において，節 0，2 の実行パスでは，式 exp が 2 回実行されるので冗長であるが，節 1，2 の実行パスでは冗長でないので，節 2 の exp は部分冗長である．節 2 の式を単純に除去してしまうと，節 1，2 の実行パス上で式 exp が計算されなくなり，プログラムの意味が変わってしまう．部分冗長な式は，先行して同じ値を計算する式が存在しない実行パス上に式を挿入することで，全冗長な式に変換することができる．図 6 (a) の節 1 に exp を挿入すると，すべての実行パスで式が冗長な全冗長となり，図 6 (b) のように冗長性を除去できる．

さらに PRE は**ループ不変コード**（loop-invariant code）[2)] をループの外へ移動させる能力を持つ．

図 7 (a) において，節 0 は，**ループ前ヘッダ**（loop pre-header）[2)] と呼ばれ，ループ外部からの複数の辺をまとめて，ループ本体に入る辺を 1 つにする節である．**ループ前ヘッダ**（loop pre-header）[2)] である節 0 に式 a+b を挿入すると，ループ内部の式 a+b は全冗長となる．すなわち，ループ内部の式 a+b は部分冗長な式であり，PRE の式の挿入と冗長な式の除去によって，図 7 (b) のように，ループ不変コードを移動させることができる．

以上のように，PRE は式の挿入をともなって，部分冗長を全冗長に変換し，冗長な式を除去する．この式の挿入と除去によるプログラム変形は，コード巻上



(a) (b)

図 7 ループ不変コードの移動
Fig. 7 Loop-invariant code motion.

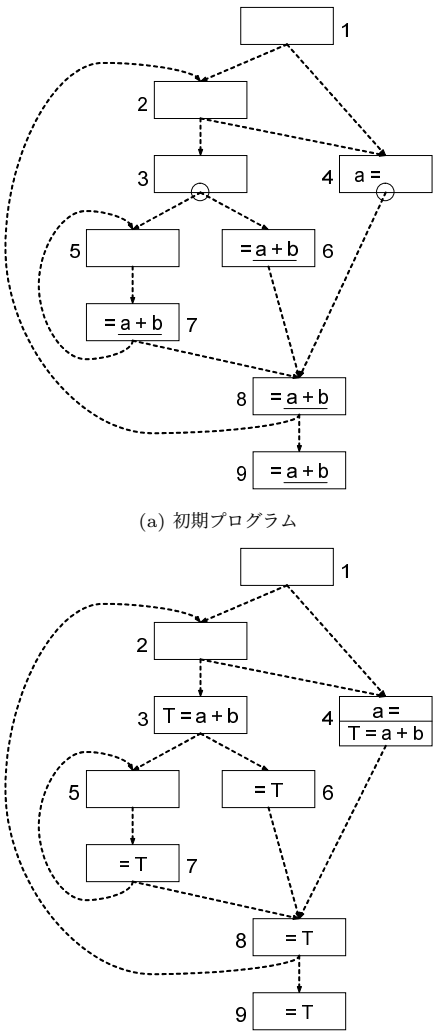
げに相当する．

PRE では，除去可能な式の出現と挿入点を決定するために，データフロー解析を用いる．本手法では特に，式の巻上げを可能な限りプログラムの終了点に近い位置へ行う，**なまけたコード移動**（Lazy Code Motion，以降 LCM）[9] と呼ばれるデータフロー解析を用いる．PRE では，式の値を保存するために一時変数の導入が必要になるが，LCM は一時変数の生存期間を最も短くする挿入を行う．変数の生存期間が短くなると，変数をレジスタに割り付ける機会を増す可能性がある．さらに，PRE は次の 2 つの性質を持つ．

(1) 実行パス上で計算される式の数を増やさない．
(2) 冗長な式を除去する効果のない無用な巻上げを行わない．

本研究では，Paleri らによって提案された LCM アルゴリズム[21] を利用する．彼らのアルゴリズムでは，1 つの文を CFG 節としているので，以降，CFG 節は文を表すものとする．データフロー解析の結果は，補償コードの挿入点を表す $INSERT$ と，冗長な式を一時変数で置き換えて除去することを表す $REPLACE$ という，2 つの述語の真偽値によって得ることができる．

PRE によって冗長な式を除去する過程を，**図 8** に示す．PRE の対象となる式は a+b である．図 8 (a) において，丸印で示したプログラム点では，$INSERT = true$ であり，下線の式では，$REPLACE = true$ である．この結果に基づいてプログラムを変形すると，図 8 (b) の結果を得る☆．



(a) 初期プログラム

(b) 冗長除去後

図 8 部分冗長除去の例
Fig. 8 Example of PRE.

5. 部分冗長除去に基づく大域命令スケジューリング

前述したように，本手法では，命令スケジューリングアルゴリズムの一部として PRE を組み込む．この PRE に基づく命令スケジューリングは，単に投機的命令スケジューリングで生じた冗長性を除去するだけでなく，次の性質を持つ．

(1) 最適化効果のないスケジューリングを行わない．
(2) 命令をスケジュールする過程で，自動的に，補償コードを最適なプログラム点に挿入する．
(3) 任意のループ構造に対して，ループスケジューリングの 1 つである，ループシフティングを行う．

本章では，本手法のアルゴリズムを示したのち，(1) と (2) の性質について，アルゴリズムを基に説明する．性質 (3) については，次章で述べる．

5.1 アルゴリズム

命令スケジューリングは空きリソースに命令をスケ

☆ いくつかの PRE では，クリティカルエッジである辺 (3, 5) を分割する形で新たに導入する節と，節 6 の入口へ補償コードの挿入を行うが，Paleri らのアルゴリズムでは新たな節の導入を抑制するために，節 3 の出口に補償コードを挿入する．

ジュールすることによって，実行パスの長さを短くするコード最適化なので，空きリソースとスケジュールする命令の選び方によって，いくつかの戦略が存在する．本手法は，これらの戦略に依存しないので，空きリソースとスケジュールする命令は適切な方法で決定されていると仮定する．なお，本手法を実装する際に採用した戦略は，7 章で述べる．

本アルゴリズムの手順は次のとおりである．

1 空きリソース R，移動可能な式 E の組（R，E）を決定する．

2 空きリソース R へ式 E のコピーを仮にスケジュールする（以降，仮スケジュールするという）．

3 PRE のデータフロー解析を行う．

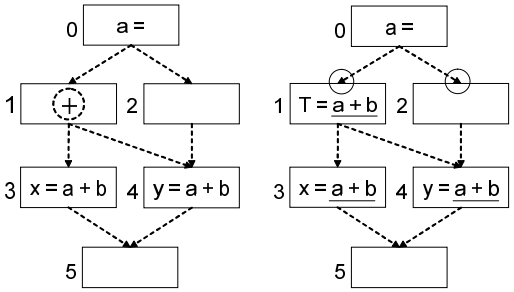
4 データフロー解析の結果，コピー元が冗長で，かつ，コピー先が冗長でないならば，PRE の挿入と除去を行い，プログラムを変形する．

5 手順 **4** の条件が成り立たなければ，仮スケジューリングを取り消し，プログラムを変形しない．

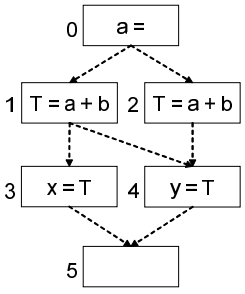
手順 **2** の仮スケジューリングが，有効であるかどうかを，手順 **4** の条件から判定し，有効な場合，PRE のデータフロー解析結果に基づいて，式の巻上げを行う．巻上げの際挿入する補償コードは，さらなる命令スケジューリングの適用によって，空きリソースに割り当てられる可能性がある．

図 9 に具体的なアルゴリズムの動作を示す．図 9 (a) の節 1 に空きリソース，節 3 に移動可能な式 `a+b` があると考える．手順 2 の仮スケジューリングで，式を空きリソースにコピーすると，図 9 (b) のようになる．この際，コピー先の式の値を保持する一時変数 `T` を導入して，元の命令を `T` と置き換えている．仮スケジューリングは空きリソースに対して行われるので，仮スケジュールする命令が実行パスを長くすることはない．

図 9 (b) において，節 3 の式は冗長であり，節 1 の式は冗長ではない．一方，節 4 の式は実行パス 0，1，4，5 で冗長であるが，実行パス 0，2，4，5 では冗長でないので，部分冗長である．ここで手順 4 の，データフロー解析を行うと，丸印で示したプログラム点が式の挿入点（$INSERT = true$）になり，下線で示した式が，除去される冗長な式（$REPLACE = true$）になる．データフロー解析の結果に基づいて，補償コードの挿入と，冗長な式の除去を行うと，図 9 (c) の結果を得る．この際，節 3 の，仮スケジュールした元の式が除去できるので，節 3 から節 1 の空きリソースに対する投機的命令スケジューリングが成功する．補償コードの挿入の際に必要になる一時変数は，新たに導入せず，すでに投機的命令スケジューリングで導入した一時変数 T を用いる．



(a) 初期プログラム (b) 仮スケジューリング



(c) PRE 適用後

図 9 PRE に基づく大域命令スケジューリング
Fig. 9 Global instruction scheduling based on PRE.

5.2 スケジューリングの条件

本手法では，命令を仮スケジュールしたのち，PRE のデータフロー解析を行い，その結果得られるコピー元とコピー先の冗長性に基づいて，実際にプログラムの変形を行うかどうかを判定する．仮スケジューリングの有効性は，手順 **4** の「コピー元が冗長，かつコピー先が冗長でない」ことによって判定できる．コピー元とコピー先の冗長性は，データフロー解析の結果に基づいて，次のように決定できる．

- **コピー元が冗長**：ある節の式が置換え可能で（$REPLACE = true$），かつ同じ節の入口に式を挿入しない（$INSERT = false$）．
- **コピー先が冗長でない**：ある節の式が置換え可能（$REPLACE = true$）で，かつ同じ節の入口に式を挿入する（$INSERT = true$）．

以降，手順 **4** の条件を**スケジューリング有効条件**と呼ぶ．

スケジューリング有効条件において，コピー元が冗長であるとは，コピー元に到達可能な，同じ値を計算する式が存在し，コピー元の式の出現が置き換えられて除去されることを意味する．また，コピー先が冗長でないとは，仮スケジュールした式が除去されずに

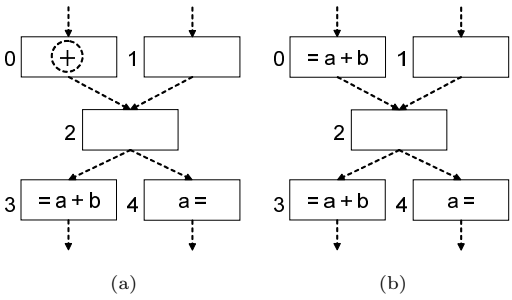


図 10 仮スケジューリングの取消し
Fig. 10 Cancel temporary scheduling.

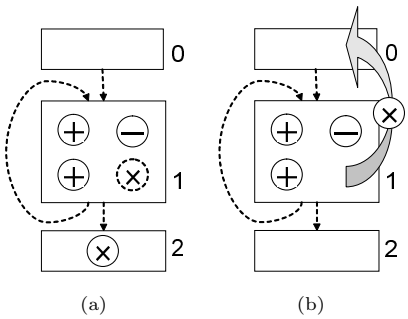


図 11 ループ内へのスケジューリングの回避
Fig. 11 Avoid scheduling into loops.

残り，その計算結果が一時変数に保持されて後続の冗長な式の出現を置き換えることを意味する．すなわち，コピー元が除去され，コピー先が除去されない状況は，コピー元位置からコピー先位置への投機的命令スケジューリングが成功したことに相当する．前述の図 9 (b) は，スケジューリング有効条件が成立する例であり，実際にプログラム変形を行うことで，投機的命令スケジューリングが実現する．

スケジューリング有効条件が成立しない場合の例を**図 10** (a) に示す．節 3 の式 `a+b` を，節 0 の＋の空きリソースに仮スケジュールすると，図 10 (b) のようになる．この図 10 (b) に PRE のデータフロー解析を適用すると，仮スケジュールする式のコピー元（節 3）が冗長にならないので，スケジューリング有効条件が成立しない．このような場合，本手法は，仮スケジューリングを取り消す．仮に節 1 に補償コードを挿入したとすると，プログラムの意味を変えずに節 3 の式を除去することができるが，実行パス 1，2，4 上で計算する式の数を増やすことになる．本手法では，PRE の性質によってこのような変形を避けることができる．

参考にスケジューリング有効条件が成立しない場合について内訳を示す．

(1) 仮スケジュールした命令自体が冗長である「コピー先が冗長」な場合
(2) 仮スケジュール先・元の間にオペランドの再定義があり，仮スケジュールした命令が後続の冗長性を取り除けない「コピー元が冗長でない」場合
(3) 補償コードが実行パスを長くする場合
(4) およびその組合せ

上記原因の，本実装における発生回数は，8 章で参考として示す．

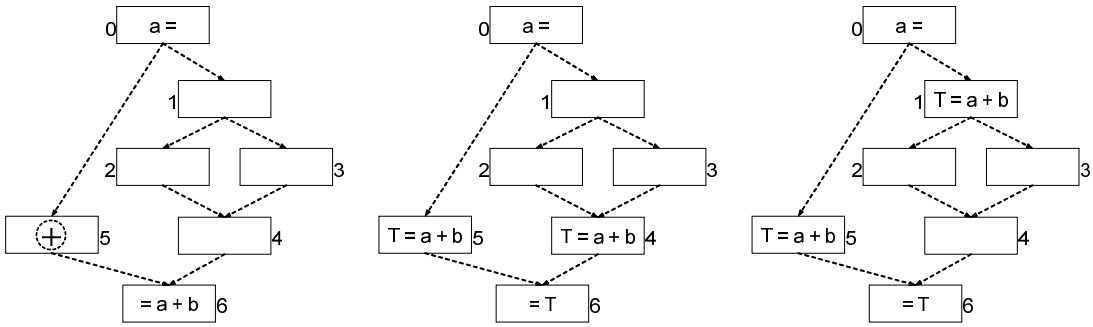
5.3 ループ内への有害なスケジューリングの回避

図 11 (a) で，空きリソース R，移動可能な式 E の組 (R，E) として（節 1 の命令 × の空きリソース，節 2 の式 ×）を考える．節 1 はループ内にあるので，投機的に仮スケジュールした命令 × はループ内でループ不変コードになる．PRE はループ不変式コードの移動の効果を含んでいるので，アルゴリズムの手順 **3** で行う PRE のデータフロー解析は，図 11 (b) で仮スケジュールした命令を，ループ不変コードとしてループ外に移動する結果を生じる．このような場合，仮スケジュールした式のコピー先が冗長となり，スケジューリング有効条件が成立しない．結果として，手順 **5** で仮スケジューリングを取り消し，ループ外からループ内への有害な命令スケジューリングを防ぐことができる．

5.4 補償コードの挿入位置

PRE は，式の巻上げによって冗長な式を除去する．この巻上げの際に行う式の挿入は，命令スケジューリングにおいて，プログラムの意味を保存するために挿入する補償コードに相当する．この補償コードは，仮スケジューリングで空きリソースに割り当てる式と違い，必ずしも空きリソースに割り当てられるとは限らない．ここで，PRE が生成する補償コードは，巻上げ可能なプログラム点のうち，最もプログラム出口に近いプログラム点に挿入される．この PRE の性質は，一時変数の生存期間を短くし，一時変数がレジスタに割り付けられる可能性を高めるという本来の効果ばかりでなく，さらなるスケジューリングの機会を減じないという効果を持つ．これらの意味で，本手法は補償コードを最適なプログラム点に挿入する．

図 12 において，補償コードを，出口に最も近いプログラム点に挿入した場合（図 12 (b)）と入口に最も近いプログラム点に挿入した場合（図 12 (c)）を示す．命令スケジューリングを行う際，空きリソースを探す領域が広いほうが，スケジューリングが成功する可能性は高まる．



(a) 初期プログラム (b) 出口に近い挿入点 (c) 入口に近い挿入点

図 12 補償コード挿入点の違い
Fig. 12 Different insertion points of compensation code.

6. ループシフティング

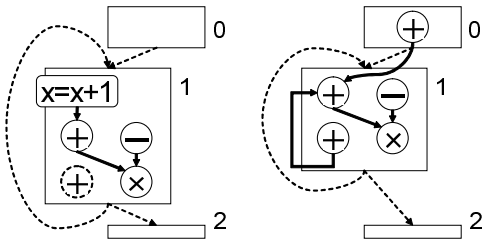
ループスケジューリングの 1 つに，**ループシフティング**（loop shifting）[17] がある．ループシフティングは，ループ内に依存を持つ命令を，ループの戻辺（back edge）に沿って，スケジュールすることによって，ループ本体の実行パスを短くする手法である．ループスケジューリングはループ専用の命令スケジューリングなので，CFG を解析してループ構造を抽出する必要があり，適用可能なループ構造にも制限があった．

本手法は，大域命令スケジューリングの枠組みの中で，ループシフティングを実現することができる．

図 13 に本手法によるループシフティングの例を示す．図 13 (a) は，ループ内の依存によって，ループ本体の実行パスをこれ以上短くできない．今，節 1 の最後に命令＋の空きリソースがあるので，空きリソース R，移動可能な式 E の組（R，E）として（節 1 の命令＋の空きリソース，命令 `x=x+1` 中の式 `x+1`）を考える．

ここで，図 13 (b) が示すように，候補の命令を戻辺に沿って空きリソースにスケジュールすると，ループ本体の実行パスを短くできる．この際，節 0 の**ループ前ヘッダ**（pre-header）に，補償コードを挿入する必要がある．

本手法にとって，ループシフティングは，スケジューリング対象の式と空きリソースが同一ループ内に存在する特殊な場合にすぎない．これは本手法が，ループを特別扱いすることなく，ループシフティングと同等な効果を得ることができることを示している．また，ループシフティングは，ループ前ヘッダを持つループ構造を前提とするが，本手法は任意の制御構造に対して適用可能なので，任意のループ構造に対するループシフティングを実現できる．



(a) (b)

図 13 ループに対する本手法の適用例
Fig. 13 Example of proposal technique applied into loop.

次に多重ループに対する，本手法のループシフティングについて考える．**図 14** (a) で空きリソース R，移動可能な式 E の組（R，E）として（節 2 の命令＋の空きリソース，節 2 の命令 `x=x+1` 中の式 `x+1`）を考える．通常のループシフティングによる効果は図 14 (b) のようになる．ここで，ループシフティングの補償コードとして節 1 へコピーされた命令が，外側のループに対してループ不変コードになっていると仮定する．この場合，本手法を適用すると，PRE のループ不変コード移動の性質から，図 14 (c) の結果を得る．

このように，本手法を多重ループに適用すると，ループシフティングの補償コードを，ループの外あるいは，より外側のループに挿入することができる．

7. 実　　装

本研究では，本手法の実現性を確認し，その効果を示すために，PRE に基づく大域命令スケジューラのプロトタイプを作成した．実装には **COINS コンパイラ・インフラストラクチャ**（以降 COINS）[3] を用いた．本章では，実装における重要な点について説明する．

COINS は，複数言語，複数機種を扱えるコンパイラ

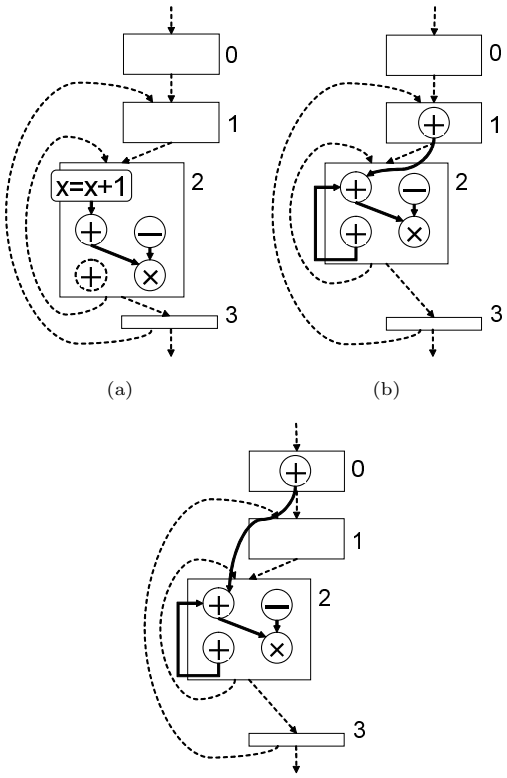


図 14 多重ループに対する本手法の適用例
Fig. 14 Example of proposal technique applied into nested loop.

の共通基盤であり，種々の手続き型言語に共通な**高水準中間表現**（High-level Intermediate Representation，以降 HIR）と，様々な機械語を抽象化した**低水準中間表現**（Low-level Intermediate Representation，以降 LIR）を持つ．HIR と LIR は，次のモジュールの組合せや変更を実現する．

(1) ソース言語から HIR への変換部
(2) HIR に適用する各種最適化部
(3) HIR から LIR への変換部
(4) LIR に適用する各種最適化部
(5) LIR から機械語への変換部

図 15 に COINS の基本的な処理フローと中間表現との関係を示す．HIR の処理を行う部分をフロントエンド，LIR の処理を行う部分をバックエンドと呼ぶ．COINS バックエンド[11] は，リターゲット型コード生成機能を持ち，Target Machine Description（TMD）と呼ばれる LIR パターンと機械語の対応記述を基に，LIR とのパターンマッチングを行う．COINS が備える命令スケジューラは，LIR レベルで命令の並べ替えを行うので，機械語非依存のスケジューラである．

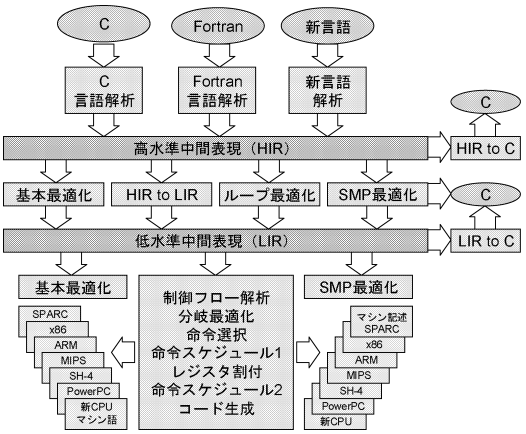


図 15 COINS コンパイラ・インフラストラクチャ
Fig. 15 COINS compiler infra structure.

7.1 本手法を行うタイミング

命令スケジューリングは，レジスタ割付けの前後で行われるのが一般的である．現在の COINS バックエンドのレジスタ割付けアルゴリズムが，できるだけ少ないレジスタを使う戦略なので，演算の途中結果を含むいくつかの変数は同じレジスタに割り付けられる傾向がある．その結果，命令間の依存関係が複雑化し，ほとんど並列実行することができなくなる可能性がある．この理由で，本手法はレジスタ割付け前に行った．

7.2 制御フローグラフのたどり方

本手法が挿入する補償コードは，PRE の性質から，最もプログラムの終了点に近いプログラム点に挿入される．補償コードはさらなる命令スケジューリングによって並列化できる可能性があるので，プログラムの開始点に近い基本ブロックから順番に，スケジューリングを適用するのが望ましい．本実装では，CFG の節をトポロジカルソート順序[7] でたどり，後続節は必ず先行節の後にスケジュールされるようにした．

7.3 空きリソースの検出

空きリソースの検出は，COINS が標準で備える命令スケジューラを改良して実現した．この命令スケジューラは SPARC[6),20)] をターゲットにした，リストスケジューリング[12),13),15),23)] を行う．SPARC は RISC アーキテクチャを採用しているので，パイプラインを抽象化しやすく，コンパイラが利用できるレジスタの数も，十分な数用意されている．基本的にプログラムは順番に実行されるが，依存関係を保ち，資源競合しない場合，パイプライン処理によって並列に実行できる．

本実装では，COINS の命令スケジューラ[3] 同様，スケジュールする命令が依存する先行命令の完了待ちを

```
READY を初期化; // 先行節がない節の集合;
while (READY が空でない) {
    A = READY の中の優先度の最も高い節を取得;
    if (現在のサイクル C に A はスケジュールできない
        && A が READY の唯一の要素) {
        空きリソースを検出;
        // 本手法はここに実装
    } else {
        A をスケジュール;
        データ依存グラフと READY から A を除去;
        A に依存する命令のレイテンシーを反映;
        先行節を持たない節を READY に追加;
        サイクル C を進める
    }
}
```

図 16 空きリソースの検出
Fig. 16 Detect idle resource.

している箇所に，空きリソースが存在すると仮定している．

図 16 に，空きリソース検出のアルゴリズムを簡潔に示す．命令 A をスケジュールできる時刻が現在のサイクルよりも後の場合，待ちが発生し，そのサイクルに空きリソースがあると考える．

7.4 移動する式の選択

移動する式を探すために走査する基本ブロックは，空きリソースを持つ基本ブロックの直接の後続ブロックとする．各後続ブロックのデータ依存グラフを作り，初期状態で依存する命令がない命令を 1 次候補とする．すべての後続ブロックの 1 次候補のうち依存関係パスの命令レイテンシの合計が最大のものをスケジュールする候補とする．

この戦略は非常に簡易なものであるので，改善策は 9.2 節で述べる．

8. 評　　価

COINS に実装したプロトタイプを用いて実験を行い，本手法の最適化効果を検証した．実験環境と，最適化効果の比較のために利用したコンパイルオプションを，それぞれ**表 1** と**表 2** に示す．COINS の最適化オプション O2 は，必ず実行される HIR レベルの基本最適化に加え，SSA 最適化を適用する．SSA 最適化は，LIR に対して行われ，SSA 変換→ 3 番地コードへの変換→ CSE →定数伝播と畳込み→ループ不変コード移動→帰納変数の演算の強さの軽減と判定の置換え→ループ不変コード移動→定数伝播と畳込み→コピー伝播→ PRE →定数伝播と畳込み→無用コード除去→ SSA 逆変換，を行う[3),10)]．さらに，整数の積算/除算の計算にプロセッサの命令を利用するオプションを有効にする．旧世代の SPARC プロセッサは，整数の積算/除算の計算にライブラリ関数を用いるが，関数呼出しは，メモリの曖昧性の問題から，本手法による命令の移動範囲を狭くする可能性がある．

表 1 実験環境
Table 1 Exprimental environment.

マシン名	Sun Blade 150
プロセッサ	UltraSPARC-IIe 550 MHz×1
2 次キャッシュ	512 KB
メモリ	1152MB
ハードディスク	30 GB
OS	Solaris 10
Java 仮想マシン	Ver. 1.5.0_06-b05
COINS	Ver. 1.4.1.1_070120

表 2 最適化オプション一覧
Table 2 List of optimize options.

coins	最適化オプションなし
coins O2	O2 オプション
coins O2+sched	O2+List scheduling（COINS 標準）
coins O2+new	O2+本手法

評価に用いたベンチマークは大きく分けて，小さなプログラム，大きなプログラムの 2 種類である．それぞれの結果を，**図 17** と**図 18** に示す．また詳細な実行時間の数値を，**表 3** に示す．表 3 の一番下の段は，投機的命令スケジューリングを行わない通常の命令スケジューラを適用した場合を基準として，本手法によってどれだけ高速化したかを示している．

小さなプログラムで構成されたベンチマークは次のような特徴を持つ．

- **heap**：ヒープソートを行う．ソートアルゴリズムは再帰呼出しを使わずに実現されている．プログラムの大部分は比較演算やロード/ストア命令で構成されている．
- **komachi**：ループだけで構成された小町算を行う．入れ子ループの内側で整数の積算命令を行う．
- **matmult**：500×500 の行列 3 つの積を計算する．浮動少数点の積算命令を行う．
- **queen**：再帰呼出しを用いて n クイーン問題を解く．ループ内部を含めて，プログラムの大部分はロード/ストア命令で構成されている．
- **soukan**：相関係数を求める．倍精度の 20 万の要素を持つ配列に対して，ロード/ストア，四則演算を 500×20 万回繰返す．

図 17 の結果から，すべてのプログラムで，本手法によるスケジューリングを適用した場合の実行時間が少なくなっている．最適化効果が最も高い komachi では，従来の命令スケジューラとの性能比較において，6.5%の改善が見られた．queen は，通常の命令スケ

表 3 実行時間詳細（秒）
Table 3 Detailed execution time (sec).

	heap	komachi	matmult	queen	soukan	gzip	mcf	bzip2	art	equake
coins	25.00	172.02	11.88	11.79	79.36	1235	1428	1299	2377	1397
coins O2	23.55	144.74	7.75	9.19	51.77	1101	1540	1106	2296	1209
coins O2+sched	23.42	141.71	7.30	12.08	47.46	978	1502	993	2225	1071
coins O2+new	23.44	132.45	7.16	8.54	45.22	955	1395	992	2204	1071
speed-up(%)	-0.09	+6.54	+2.01	+29.35	+4.72	+2.35	+7.12	+0.1	+0.94	0

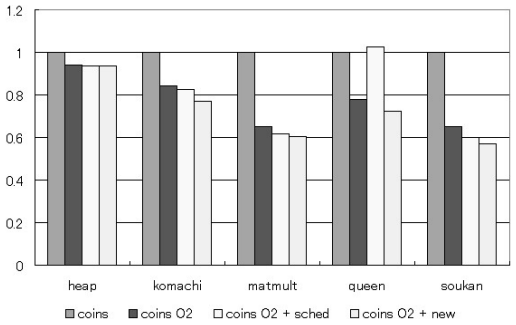


図 17 実行時間（小さなプログラム）
Fig. 17 Execution time (small programs).

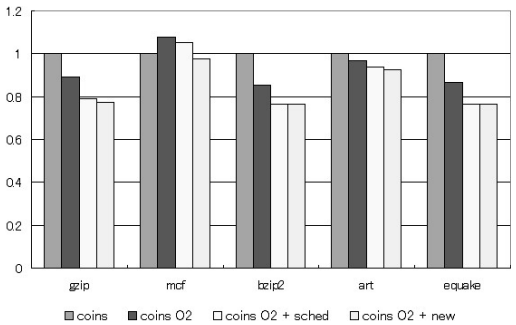


図 18 実行時間（大きなプログラム）
Fig. 18 Execution time (large programs).

ジューラによる最適化が明らかに失敗しているので，改善率は評価できないが，本手法の結果は O2 オプションの結果よりも優れているで，本手法による最適化効果が認められる．heap では，本手法の最適化効果が見られないが，従来の命令スケジューラとほぼ同じ結果になっている．これは，本手法の命令スケジューリングが持つ，実行パスを長くしないという特徴によると考えられる．

次に，大きいプログラムとして，SPEC2000 ベンチマーク[19] のうち，整数演算系 gzip，mcf，bzip2，浮動小数演算系 art，equake を評価した．

図 18 の結果から，小さなプログラムでの傾向と同様に，大きなプログラムに対しても，本手法は最適化効果が得られるか，少なくとも悪影響を与えないことが分かる．通常の命令スケジューラとの性能比較については，gzip や mcf では数%の改善が見られ，bzip2，equake ではほぼ同じ結果になった．mcf に関しては，O2 オプションによる最適化が失敗しているので，単純に 7%の改善があるとはいえないが，本手法による一定の効果を示している．

さらに，参考として，プロトタイプの実験から得た，本手法の試行回数に関連するデータを**表 4** と**表 5** に示す．まず，上の段から順に，通常のリストスケジューリングを行った命令数，空きリソースに対して仮スケジュールを行った回数，本手法の成功回数を示している．続いて，本手法のスケジューリング有効条件を成立させない,「コピー先が冗長」・「コピー元が冗長でない」という判定に加えて，本実装では取り扱うことのできない，クリティカルエッジ上への挿入に相当する「辺上への挿入が発生する」という失敗条件の判定回数もあわせて示している．最後の「コンパイル時間増加率」は，COINS 標準の命令スケジューラに対する，本手法のコンパイル時間増加率を示している．なお，プログラム中の文の数（節の数）を N，空きリソースの数を R とすると，本手法のアルゴリズムの最悪のオーダは，$O(NR)$ 程度と考えられる．

9. 今後の課題

9.1 コピー代入の影響

本手法によって，投機的に命令をスケジュールした後，元の命令があった場所は，空きリソースになるのではなく，コピー代入が残る．このコピー代入は，他の命令の投機的なスケジューリングを阻害する可能性がある．

```
a = b*c;
d = a*c;
```

上のような命令列で，式 `b*c` を本手法によって巻き上げると，下に示すように，新たに一次変数が導入され，コピー代入が生じる．コピー代入は，依存関係を保持するので，後続の式 `a*c` を本手法のスケジュール候補に含めることができない．

```
a = T;
```

表 4 スケジューリングの分類とコンパイル時間（小さいプログラム）
Table 4 Classification of scheduling and compilation time (small programs).

	heap	komachi	matmult	queen	soukan
通常スケジューリング	422	196	234	300	372
仮スケジューリング	37	9	54	28	10
本手法成功	7(19%)	4(44%)	9(17%)	5(18%)	10(100%)
コピー先冗長	30	5	45	16	0
コピー元冗長でない	28	0	41	16	0
辺上へ挿入	0	2	0	7	0
コンパイル時間増加率	111%	108%	113.3%	109.7%	109.9%

表 5 スケジューリングの分類とコンパイル時間（大きなプログラム）
Table 5 Classification of scheduling and compilation time (large programs).

	gzip	mcf	bzip2	art	equake
通常スケジューリング	11689	2381	6461	3096	4137
仮スケジューリング	954	341	625	124	626
本手法成功	297(31%)	56(16%)	159(25%)	13(11%)	16(3%)
コピー先冗長	486	251	361	103	592
コピー元冗長でない	359	221	190	82	525
辺上へ挿入	211	54	142	17	105
コンパイル時間増加率	148%	169%	422%	491%	763%

```
d = a*c;
```

このような問題は，コピー伝播を適用することによって緩和することができる．1 つの命令に対して本手法が成功するたびに，生じるコピー代入に対してコピー伝播を行うと，コピー代入との間の依存関係が消え，後続の式を，新たに本手法の候補することができる．

```
a = T;
d = T*c;
```

コピー伝播を行った結果，左辺の変数が使われなくなったコピー代入は，**無用コード除去**（dead code elimination）[2),8)] によって除去できる．結果として，コピー代入を実行するために必要なリソースが開放されるので，空きリソースが生じ，さらなるスケジューリングが可能になる．

9.2 解析範囲の拡張とパスの限定

現在，空きリソースを発見した場合，空きリソースがある基本ブロックの直接の後続ブロックだけが，スケジュールする命令を探す候補になる．さらに後続ブロックをたどって，式を探す候補のブロック数を増やせば，本手法を適用できる機会が増えると考えられる．また，基本ブロック間のデータ依存関係も考慮できるような大域的なデータ依存グラフを利用することで，データ依存のクリティカルパスを正確に計算でき，スケジューリング効果の高い命令を候補にできる可能性があると考える．さらに，実行パスの実行頻度をプロファイリングによって取得し，CFG のクリティカルパス上の命令を優先する方法もある．

投機的命令スケジューリングは，移動する命令のレイテンシが大きく，実行される頻度が高いほど効果的であるので，前述したような方法は有効であると考えられる．

9.3 COINS の改良

アルゴリズムだけではなく，COINS にも改良を加えることで，さらに本手法の効果の向上が可能であると考えられる．

9.3.1 メモリの曖昧性の除去

別名解析や手続き間解析を備えると，メモリの曖昧性から生じる依存を減らし，PRE のデータフロー解析や，投機的命令スケジューリングによって命令を移動する範囲を広げることができる可能性がある．

9.3.2 LIR と機械語の 1 対 1 対応

COINS が備える命令スケジューラは，機械語に依存しない LIR レベルで命令の並べ替えを行うが，必ずしも LIR の 1 命令に 1 つの機械語が対応するわけではない．たとえば，LIR の条件分岐を表す 1 命令は，SPARC マシンでは機械語の 3 命令（比較命令，分岐命令，遅延分岐のための nop 命令）に対応する．この問題は，次の点で，本手法の効果を阻害する要因になる可能性がある．

- 1 つの LIR が対応する複数の命令間では命令スケジューリングができない．複数の塊として意味を持つ機械語間には，依存関係がある場合が多いので，そこに命令レイテンシによる多くの空きリソースが存在するとしても，検出することはできない．
- 1 つの空きリソースに対して，複数の機械語に展

開される LIR を投機的にスケジュールした場合，展開される複数の機械語間に存在する依存関係によって，パイプライン処理のスループットが下がる可能性がある．

LIR の 1 命令と機械語を 1 対 1 に対応させることができれば，上記のような問題が緩和され，さらなる最適化効果の向上が可能になると考えられる．また，パイプラインにおける詳細な資源制約を記述することもできるようになると考えられる[5),22)]．

9.4 動的 CSE との比較

8 章では，COINS が備える命令スケジューラ（リストスケジューリング）と，本手法を実装した命令スケジューラとの比較実験を行い，結果から，本手法の最適化効果が認められることを示した．今後の課題として，命令スケジューリングと冗長除去の単純な組合せである動的 CSE と比べて，本手法が，どの程度多くの冗長性を除去できているのか，動的 CSE との比較実験を行う必要がある．

10. 関 連 研 究

ループ不変コード移動[2)] は，Aho らによって，基本的なアルゴリズムが示されている．この基本的なループ不変コード移動は，不変コードをループ前ヘッダに移す方法であり，1 つの使用に対して，複数の定義が到達する操作を巻き上げることができない．

CSE[2)] は，先行するプログラム点で計算済みの式の値を一時変数に保持しておき，後続の共通部分式を一時変数で置き換えて，冗長な式を除去する手法である．ここで，CSE が対象とするのは，全冗長な式であり，部分冗長な式を除去することはできない．

PRE は，コード巻上げを行って部分冗長を除去する手法であり，ループ不変コード移動の効果も含む．Morel らによって初めて示された PRE[4)] は，無用な巻上げを行う欠点を持ち，さらに双方向データフロー解析を用いるので計算が複雑であった．Knoop らは，単方向データフロー解析の組合せで PRE を実現できるように改良し，無用な巻上げを行わない LCM を提案した[9)]．これらの PRE のうち，本手法では，無用な巻上げを行わない方法を採用している．LCM は，補償コードをできる限りプログラムの終了点に近いプログラム点に挿入するので，さらにスケジューリングを適用する場合に，補償コードを空きリソースに割り当てる機会を減じない．現在使用されている PRE のほとんどは，無用な巻上げを行わない手法である．

命令スケジューリングに初めて冗長除去を導入したのは，S. Gupta らの動的 CSE[16)] である．動的 CSE は，投機的命令スケジューリング[18)] を行うたびに，CSE を適用し，生じた共通部分式を除去する．しかし，CSE を用いる動的 CSE では部分冗長を除去することができない．これに対して，R. Gupta は，命令スケジューリングで生じた冗長性を，PRE で除去する手法を提案した[14)]．この手法はループ構造を特別扱いして，最適化対象から除外する必要があるので，ループスケジューリングを行うことができず，プログラム全体に対して統一的な最適化を行うことができない．本手法は，PRE を命令スケジューリングの一部として統合することで，命令スケジューリング自体が，任意のプログラムに対する補償コードの挿入や，ループ構造の認識を必要としないループシフティングを可能にする．すなわち，本手法は大域命令スケジューリングとループスケジューリングを統一的に扱う一般性を有しているといえる．

11. ま と め

本研究では，部分冗長除去に基づく大域命令スケジューリングの提案と，従来手法との効果の比較を行った．本手法は，PRE を命令スケジューリングの一部として組み込んでいるので，大域命令スケジューリングで問題となる補償コードを，自動的に最も最適なプログラム点に挿入する．さらに，命令スケジューリングの結果に関して，PRE と同様に実行パスを長くしないことを保証する．

本手法の最適化効果を検証するために，COINS コンパイラ・インフラストラクチャ上で本手法のプロトタイプを実装し，評価を行った．その結果，SSA 最適化とリストスケジューリングを含む多くの最適化を適用した状態から，さらに数%の性能向上を達成した．これは，本手法が持つ，ループシフティングを含む，投機的命令スケジューリングの命令レイテンシの隠蔽効果によるものと考えられる．実行結果が悪くならないことについても，PRE を命令スケジューリングの一部として統合する本手法のアルゴリズムが，PRE の特徴を備えていることを示している．

謝辞 本研究の一部は，科学研究費補助金の補助に基づいている．

参 考 文 献

1) 安藤秀樹：命令レベル並列処理，コロナ社 (2005).
2) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compiler: Principles, Techniques and Tools*, Addison-Wesley (1986).

3) COINS Project. http://www.coins-project.org
4) Morel, E. and Renvoise, C.: Global Optimization by Suppression of Partial Redundancies, *CACM*, Vol.22, No.2, pp.96–103 (1979).
5) Free Software Foundation, Inc: GNU C Compiler Internals. http://gcc.gnu.org/onlinedocs/gccint/
6) SPARC International, Inc.（著），相越克久，田中長光（訳）：SPARC アーキテクチャ・マニュアルバージョン 8，トッパン (1992).
7) 石畑 清：アルゴリズムとデータ構造，岩波書店 (2000).
8) Knoop, J., Ruthing, O. and Steffen, B.: Partial Dead Code Elimination, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.147–158 (1994).
9) Knoop, J., Ruthing, O. and Steffen, B.: *Lazy Code Motion*, pp.233–245, ACM (1995).
10) 佐々研究室：静的単一代入形式最適化システム外部仕様書，技術報告，Coins Project (2005).
11) 森公一郎：COINS バックエンド部の実装，技術報告，Coins Project (2005).
12) 中田育男：コンパイラの構成と最適化，朝倉書店 (1999).
13) Allen, R. and Kennedy, K.: *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers (2001).
14) Gupta, R.: A Code Motion Framework for Global Instruction Scheduling, *International Conference on Compiler Construction*, pp.219–233 (1998).
15) Morgan, R.: *Building an Optimizing Compiler*, Butterworth-Heinemann (1998).
16) Gupta, S., Reshadi, M., Savoiu, N., Dutt, N.D., Gupta, R. and Nicolau A.: Dynamic Common Sub-Expression Elimination during Scheduling in High-Level Synthesis, *Proc. International Symposium on System Synthesis*, pp.261–266, ACM (2002).
17) Gupta, S., Dutt, N.D., Gupta, R.K. and Nicolau, A.: Loop Shifting and Compaction for the High-level Synthesis of Designs with Complex Control Flow, *Proc. Design, Automation and Test Conference*, pp.114–121, ACM (2004).
18) Gupta, S., Gupta, R., Dutt, N.D. and Nicolau, A.: *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*, Kluwer Academic Publishers (2004).
19) Standard Performance Evaluation Corporation. http://www.spec.org/
20) Sun microsystems, Inc: UltraSPARC Processors Documentation. http://www.sun.com/processors/
21) Paleri, V.K., Srikant, Y.N. and Shankar, P.: A Simple Algorithm for Partial Redundancy Elimination, *SIGPLAN Notices*, Vol.33, No.12, pp.35–43 (1998).
22) Makarov, V.N.: The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC, *GCC Summit proceedings* (2003).
23) Srikant, Y.N. and Shankar, P.: *The Compiler Design Handbook*, CRC Press LLC (2003).

(平成 19 年 1 月 22 日受付)
(平成 19 年 5 月 18 日採録)

勝原 達也

2007 年東京理科大学大学院理工学研究科情報科学専攻修士課程修了．現在，株式会社野村総合研究所基盤ソリューション事業本部基盤ソリューション事業一部勤務．

滝本 宗宏（正会員）

1994 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．1999 年東京理科大学理工学部情報科学科助手．現在，東京理科大学理工学部情報科学科講師．2005〜2006 年米国カリフォルニア大学アーバイン校組み込み計算機システムセンター訪問研究員．工学博士．ソフトウェア工学，プログラミング言語およびその処理系に興味を持つ．ACM，IEEE，日本ソフトウェア科学会各会員．