質問伝播に基づく大域ロード命令集約

澄川 靖信[1,a)]　滝本 宗宏[1,b)]

概要：現在のプロセッサは，プロセッサの演算速度に比べて低速な主メモリと，主メモリよりも高速なキャッシュメモリを備えていることが多い．このような構成のプロセッサ上で，プログラムを効率的に動作させるためには，キャッシュメモリの有効な利用が重要である．キャッシュメモリは，プログラムの時間的局所性と空間的局所性を高めることによって有効に働くので，同じ配列へのアクセスを連続して行う場合に効果的である．著者らは，前研究において，データフロー解析を利用してプログラム全体を解析し，同じ配列へアクセスするロード命令を互いに近いプログラム点に移動させることによって，キャッシュメモリのヒット率を向上させる大域ロード命令集約を提案した．しかしながら，従来法は，ロード命令の出現ごとに，データフロー解析を行うので，解析コストが高くなる傾向にあった．本発表では，質問伝播と呼ばれる要求駆動型の解析方法を用いることによって，プログラムの解析範囲を限定し，解析効率を向上させる手法を提案する．本手法を C コンパイラ上で実装し，SPEC2000 ベンチマークを用いて評価を行った．その結果，従来法と比べて，解析効率が平均で約 63%向上することを確認した．

Global Load Instruction Aggregation Based on Query Propagation

Sumikawa Yasunobu[1,a)]　Takimoto Munehiro[1,b)]

Abstract: Most modern processors have some cache memories that are much faster than a main memory, and it is important to utilize these effectively for efficient program execution. The cache memories function well if temporal or spatial localities in the program are enhanced. Therefore, the cache efficiency can be improved by making accesses to the same array continuous. We previously proposed global load instruction aggregation (GLIA), which reorders load instructions in the way that ones accessing to the same arrays were aggregated. Because GLIA exhaustively analyzes the entire program based on dataflow analysis to each load instruction, this is costly. We propose a technique to realize the GLIA based on demand-driven analysis which is called query propagation in order to restrict the program region to be analyzed. We have implemented our technique in a real compiler, and evaluated it on SPEC benchmarks. The experimental results show that our technique can decrease analysis efficiency about 63% in average.

1. はじめに

現在のプロセッサは，レジスタ参照と比較して，アクセスが低速な主メモリと，主メモリよりも高速にアクセスできるキャッシュメモリから構成されているものが多い．

プログラムの実行中に，ある番地 x のデータが必要になれば，まずキャッシュメモリを調べる．このとき，データが，キャッシュメモリで見つかることを**キャッシュヒット**（cache hit）と呼び，見つからないことを**キャッシュミス**（cache miss）と呼ぶ．キャッシュヒットが生じれば，主メモリへのアクセスを行わないので，演算を高速に継続できる．一方，キャッシュミスが生じると，主メモリの x 番地を含めた近傍の番地からデータを読み出し，次の x の参照が，キャッシュヒットを生じるように，キャッシュメモリに配置する．この際の x 番地の参照は，大きな遅延を生じ

[1] 東京理科大学
Tokyo University of Science
[a)] yas@cs.is.noda.tus.ac.jp
[b)] mune@cs.is.noda.tus.ac.jp

ることになる．x 番地のデータのキャッシュメモリへの配置は，他の番地のデータがキャッシュメモリから除去されることを意味するので，番地の遠い主メモリへの連続したアクセスは，頻繁なキャッシュミスを生じ，プログラム全体の実行効率を低減させる可能性がある．

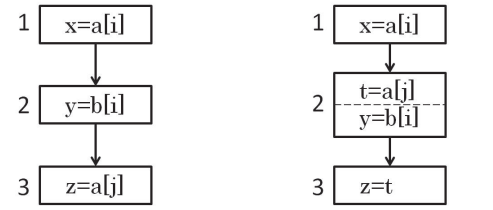
一方，主メモリからデータがロードされると，同じ配列に属する他のデータも同時にキャッシュメモリに配置される可能性が高いので，同じ配列へのアクセスが連続して行われるようにロード命令を移動させると，キャッシュヒットを生じる可能性を高めることができる．

著者らは，同じ配列へアクセスするロード命令が連続して実行されるようにプログラムを変更することによって，キャッシュミスの原因の 1 つである競合ミスを低減させる**大域ロード命令集約** [13]（Global Load Instruction Aggregation，以降 GLIA と呼ぶ）を提案した．GLIA は，**制御フローグラフ**（Control Flow Graph，以降 CFG と呼ぶ）を**トポロジカルソート順序**（toporogical sort order）で訪問し，ロード命令の出現ごとに，プログラムに含まれるすべてのロード命令のアクセス先を解析し，メモリアクセス順序をデータフロー方程式を利用することによって求め，配列へのアクセス順序が連続されるようにロード命令を移動する．したがって，GLIA は，ロード命令が出現するごとにプログラム全体を繰り返し解析するので，解析コストが高くなる傾向があった．

本研究では，質問伝播と呼ばれる要求駆動型の解析法を用いることによって，プログラムの解析範囲を限定しながら，GLIA と同じ効果を得られる手法を提案する．本手法は，ロード命令 e が出現するごとに，「e は冗長か」というクエリをプログラムの制御とは逆向きに伝播させる．クエリを伝播する際に，もし e と同じロード命令が出現したならば解 $true$ を返す．一方，e の値を変更するストア命令や，e に含まれる変数の定義の出現，クエリがプログラムの開始点に到達したときは解 $false$ を返す．クエリの解として $false$ が得られた際，節に同じ配列へアクセスするロード命令が含まれているかを検査する．同じ配列へアクセスするロード命令の出現以降，異なる配列へアクセスするロード命令を含む節が存在したとき，その節の入口に新しく e を挿入し，クエリの解として $true$ を返す．

例：**図 1** (a) のプログラムに GLIA を適用した結果のプログラムが図 1 (b) である．データフロー解析を用いた GLIA を適用した場合，3 つのロード命令ごとに，各ロード命令のアクセス先の解析，ロード命令を安全に移動できる節，ロード命令を移動させるのかどうかについて解析しなければならないので，プログラム全体を繰り返し解析する必要がある．

一方，本手法は質問伝播を用いているので，開始節に向かってクエリを伝播し，必要な情報が得られ次第解析を終了する．したがって，節 1 のロード命令 `a[i]` においての検査は，先行節が無いので，クエリの伝播は行わない．節 2 のロード命令 `b[i]` においては，節 1 だけにクエリを伝播させ，解析を終了する．∎



(a) 元のプログラム
Original code.

(b) GLIA の適用
Code after applying GLIA.

図 1 GLIA の効果
Fig. 1 Effectiveness of GLIA.

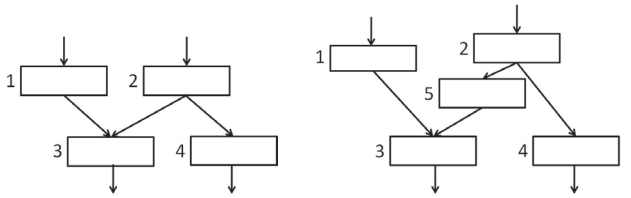
本手法の効果を確かめるために，本手法を C コンパイラ上で実装し，SPEC2000 ベンチマークで評価を行った．結果として，GLIA を適用した場合と比較すると，同程度の目的コードの実行効率を得られ，解析効率が平均で約 63%向上することを確認した．

本稿の以降の構成は次のとおりである．第 2 節で本稿で述べるプログラムの表現形式を定義する．第 3 節で GLIA の概説を行う．第 4 節で本手法が基礎とする解析方法の質問伝播を用いて，コード最適化の 1 つである部分冗長除去を実現した手法について概説する．第 5 節で質問伝播を用いて GLIA を実現する提案手法について述べ，第 6 節で実験結果とその評価を述べる．第 8 節でまとめる．

2. 入力プログラム

本手法では，適用対象のプログラムは中間表現に変換されているものとする．中間表現の各命令を文と呼び，代入文の右辺を式と呼ぶ．また，ロード命令とストア命令を配列へのアクセスを含む代入文として表す．例えば，先頭アドレスが a の配列の i 番目のデータを仮想レジスタ x へロードすることを $x = a[i]$，仮想レジスタ x を $a[i]$ へストアすることを $a[i] = x$ でそれぞれ表す．なお，本文中の説明には，理解を容易にするために，C 言語のプログラムを用いる．

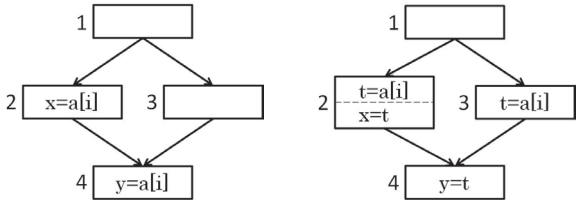
本手法では主メモリにアクセスする命令は，代入文だけに含まれると仮定する．もし代入文以外で主メモリのデータを参照する場合は，ロード命令を右辺とする代入文を新たに生成し，代入先の変数を使用する．例えば，手続き f の呼出し $f(a[i])$ は，$t = a[i]$，$f(t)$ のように複数の文に変形する．また，1 つの代入文に含まれるメモリアクセス命令は，高々 1 つになるように変形されているものとする．例えば，文 $a[i] = a[b[j]]$ は，$t = b[j]$，$t_0 = a[t]$，$a[i] = t_0$ のように複数の文に変形する．さらに，ある変数の定義と使



(a) 元のプログラム
Original code.

(b) クリティカル辺の除去
Removing critical edges.

図 2 クリティカル辺
Fig. 2 Critical edges.



(a) 元のプログラム
Original code.

(b) PRE を適用したプログラム
Code after applying PRE.

図 3 部分冗長なロード命令の除去
Fig. 3 Eliminating partial redundant load instructions.

用が同じ文では出現しないと仮定する．例えば，文 $i = a[i]$ は，$t = a[i]$，$i = t$ のように複数の文に変形する．

本手法を適用する前に，各プログラムに対して CFG が生成されているものとする．CFG は，文を節とする集合 **N**，節間の制御の流れを表す有向辺の集合 **E**，プログラムの開始点を表す特別な節 **s**，終了点を表す特別な節 **e** の 4 つ組 $(\mathbf{N}, \mathbf{E}, \mathbf{s}, \mathbf{e})$ である．また，節 n の先行節集合を $pred(n)$，後続節集合を $succ(n)$ で表す．

本手法は，コード移動に基づく手法なので，コード移動を阻害することで知られる**クリティカル辺**（critical edges）[4][8] は，取り除かれているものとする．クリティカル辺は，2 つ以上の後続節をもつ節から 2 つ以上の先行節を持つ節への辺であり，辺上に，新しい節を挿入することによって取り除くことができる．

本手法では，クリティカル辺を除去するために，2 つ以上の後続節をもつ節から出ているすべての辺上に新しく節を挿入する．

3. 大域ロード命令集約

本節で**部分冗長除去** [1] [2] [4] [5] [6] [7] [8] [9] [11] [14] [15]（Partial Redundancy Elimination，以降 PRE と呼ぶ）の基本と，PRE の実現方法の 1 つである**怠けたコード移動** [8]（Lazy Code Motion，以降 LCM と呼ぶ），および LCM を拡張して実現している GLIA について概説する．

3.1 怠けたコード移動

すべての実行経路で冗長な式は，共通部分式（common sub-expression），あるいは**全冗長**（total redundant）な式と呼ばれ，以前の計算結果を再利用することによって除去できる．一方，一部の実行経路で冗長な式は**部分冗長**（partial redundant）と呼ばれ，全冗長な式のように単純に除去できない．PRE は，部分冗長な式を，全冗長にするために新しく式を挿入し，除去する．

例：**図 3**(a) のプログラムに PRE を適用した結果が図 3(b) である．(a) の節 4 のロード命令 `a[i]` は，制御が節 2 を通るならば冗長であるが，節 3 を通るときは冗長ではない．この一部の実行経路で冗長なロード命令は，PRE によって，(b) のように変形することによって除去できる．∎

また，ループ不変式は，ループに入る実行経路上で冗長でないがループ内で冗長なので，部分冗長な式とみなすことができる．したがって，PRE ではループ構造を意識することなく，部分冗長な式の除去とループ不変式のループ外への移動を同時に行うことができる．PRE が行う式の挿入は，次の**安全性**の性質を保証することが知られている．

安全性： プログラムの任意の実行経路において，式の数を増やさない

安全性は述語 $DownSafe$ で表される**下向き安全性**（down safety）が満たされれば保証される．

$DownSafe(n)$： 節 n から終了節 **e** への任意の実行経路上の節 m に同じ式が存在し，n と m の間で式のオペランドが変更されない．

ここで，式のオペランドが変更されない節は**透過性**（transparency）をもつと言う．

PRE の実現法の 1 つである LCM は，式 e の冗長性を除去するために，2 種類の挿入節 *Earliest* と *Latest* を求める．*Earliest* 節とは，下向き安全な節の中で，最も開始節 **s** に近い節である．*Latest* 節とは，下向き安全な節の中で，*Earliest* 節と元の e の出現との間で最も e の出現に近い節である．ここで，節 n が e の出現を含むかどうかを述語 $Comp(n)$ で表す．

Earliest 節に式を挿入して冗長性を除去するアルゴリズムは，**せわしいコード移動**（Busy Code Motion，以降 BCM と呼ぶ）と呼ばれる [8]．BCM も，部分冗長性を除去することができるが，式を除去する際に，式を置き換える一時変数の生存期間を長くする傾向がある．一時変数の生存期間の伸長は，レジスタ割付においてレジスタスピルを生じる可能性を増大させるので，最適化効果を減じる可能性がある．一方，*Latest* 節へ式を挿入する LCM は，一時変数の生存期間が BCM と比べて短くなる傾向にあるので，レジスタスピルを生じる可能性を低減できる．*Latest* 節は，*Earliest* 節から，*Comp* 節を越えないように，述語

Delayed によって表される巻戻しを行うことによって得られる．最終的に，*Insert* を満たさない *Comp* 節にある式を除去する．

3.2 コード移動に基づく大域ロード命令集約

著者らは，先行研究において，先行する同じ配列へアクセスするロード命令の直後へ移動させた後，メモリアクセス順序を考慮した巻戻しを行う GLIA を提案した．GLIA は，LCM と同様に，*DownSafe*，*Earliest* 節，*Latest* 節を求め，プログラム変形を行う．

GLIA は，メモリアクセス順序を連続させるために，先行するロード命令に同じ配列へアクセスする配列参照が存在することを意味する，述語 *UpSafe* を定義する．この *UpSafe* は，PRE で用いられる上向き安全性の条件を緩和したものである．

UpSafe(n)： 開始節 **s** から節 n への任意の実行経路上の節 m にアクセス先アドレスが同じ配列参照が存在し，n と m の間で式のオペランドが変更されない．

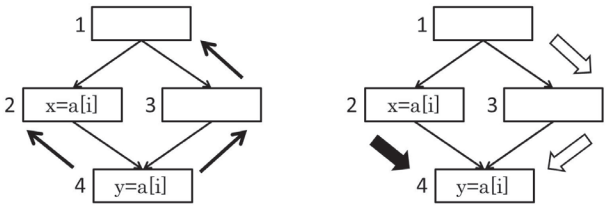
すなわち，上向き安全性を満たし，異なる配列へアクセスする配列参照を含む節の出口への巻戻しを防ぐことでメモリアクセス順序を連続させる．

4. 質問伝播に基づく部分冗長除去

PRE は，冗長な式の除去によって，その式に依存する後続の式の冗長性が明らかになる**副次的効果** [10]（second order effects）をもつことが知られている．副次的効果を反映するためには，PRE を複数回適用する必要があるので，網羅的な解析を行う一般的な PRE では解析コストが高くなる傾向があった [17].

近年，一度の適用で副次的効果を効率的に反映する手法として，**質問伝播に基づく PRE** [17]（PRE Based on Query Propagation，以降 PREQP と呼ぶ）が提案された．PREQP は，Rosen らによって提案された質問伝播と呼ばれる全冗長性を検査する手法を [12]，部分冗長性の検査を行うように拡張した手法である．PREQP は，CFG をトポロジカルソート順序で訪問し，出現した式 e に対して「e が利用可能か」というクエリを後向きへ伝播することによって冗長性を検出し，プログラム変形を行う．クエリの伝播は，e と字面が等しい式が存在すること表す解 *true*，e がその経路上では冗長でないことを表す解 *false* のいずれかが得られるまで繰り返し行われる．

もし節 n で，複数のクエリの解が得られ，*true* と *false* の両方が含まれるならば，e は n で部分冗長である．したがって，*false* を得た節が下向き安全性を満たすならば新しく式を挿入し，e が n において全冗長となるように変形する．また，クエリの式が出現した際に，新しい一時変数を用いて変形を行う．例えば，$x = e$ は，$t = e$，$x = t$ のように複数の文に変形する．



(a) クエリ伝播 Query propagation.　(b) クエリの解 Answers of query.

図 4 質問伝播に基づく解析
Fig. 4 Analysis based on query propagation.

例：図 4(a) は，図 3(a) の節 4 の配列参照 `a[i]` に PREQP を適用した際のクエリ伝播の様子を表す．節 4 から節 2 へ伝播されたクエリは，節 2 に同じ式が含まれているので，節 2 の入り口に `t=a[i]` を挿入し，元のプログラムに存在していた `x=a[i]` を `x=t` へ変形し，*true* を返す．一方，節 3 へ伝播されたクエリは，節 3 において解が得られないので，先行節へクエリを伝播させる．結果として，クエリが開始節へ伝播されたので，節 3 の解として *false* が得られる．最終的に，節 4 において，クエリの解として *true* と *false* が得られたので，*false* を得た節 3 に `t=a[i]` を挿入し，節 4 の `a[i]` を `t` で置き換える．

本稿では，説明を容易にするために，クエリの解が *true* を黒い太矢印で表し，*false* を白い太矢印で表す．∎

PREQP におけるクエリの伝播は，次の規則を上から優先して適用する．

(1) 節 n にクエリが伝播されたとき，n が開始節 **s** ならば，n におけるクエリの解は *false* である．

(2) 節 n にクエリが伝播されたとき，n にクエリと同じクエリが伝播済みならば，n におけるクエリの解は *true* である．

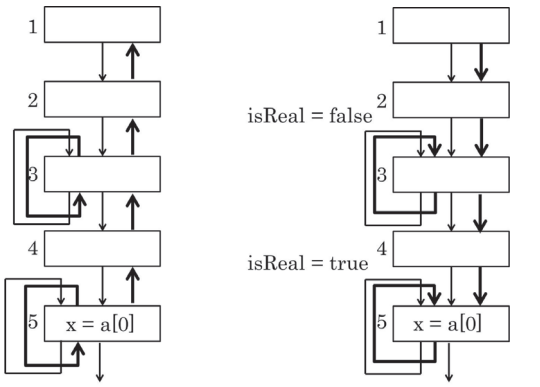
(3) 節 n にクエリが伝播されたとき，n にクエリの式が含まれているならば，n におけるクエリの解は *true* である．

(4) 節 n にクエリが伝播されたとき，n に，クエリのオペランドを変更する計算，またはクエリと同じ配列へアクセスするストア命令が含まれるならば，n におけるクエリの解は *false* である．

(5) 上記の規則のいずれにも該当しない場合，すべての先行節にクエリを伝播させる．

質問伝播は，上記の規則 (1)–(4) のいずれかが成り立つときに局所的な解が得られる．もし局所的な解が得られないときは，規則 (5) によって，クエリを先行節へ伝播する．

ここで，規則 (2) において，クエリの解を *true* とすることによって，データフロー解析の最大解を求められる．しかしながら，図 5 に示すように，規則 (2) によって不要な式の移動が行われる可能性がある．



(a) 元のプログラム Original code.　(b) クエリ伝播 Query propagation.

図 5 不要なコード移動
Fig. 5 Unnecessary code motion

例：図 5(a) の節 4 からクエリを伝播させると，節 3 からバックエッジに沿って節 3 への伝播は規則 (2) に該当するので，節 3 における解は *true* と *false* の両方が得られる．したがって，*false* を得る節 2 に新しく式を挿入すると式 `a[0]` をループの外へ移動できる．しかしながら，節 3 に式 `a[0]` は含まれないので，適切な式の挿入点は節 4 である．節 4 への挿入と比較すると，節 2 への挿入は，新しく導入する一時変数の生存期間が長くなり，レジスタ圧力を増加させる．▮

したがって，実際に式が出現したことを表す述語 *isReal* を定義し，*isReal* の *true* と，クエリの解 *false* が得られたならば，*false* を得た先行節に式を挿入する．
例：図 5 のプログラムでは，*isReal*(3) は *false* である．したがって，節 2 への挿入は行われない．一方，*isReal*(5) は *true* なので，節 4 へ新しく式を挿入する．▮

本稿で説明する PREQP は，後で GLIA への拡張を行うために，通常形式のプログラムを適用の対象としている．通常の PREQP は**静的単一代入** [7][12][16][18]（static single assignment）形式に変換されたプログラムを適用の対象としているので，本稿の PREQP ではクエリの代数変形，ϕ 関数の挿入を行う．また，本稿の PREQP は通常形式のプログラムを適用対象としているので，$Comp(n)$ が *true* となる節 n の入り口に式を挿入しなければならない．しかしながら，常に *isReal* が *true* となる節に式を挿入するとループ不変式の移動が行えないので，クエリを生成した節においては式の挿入は行わない．

上記の質問伝播は，伝播の方向を前向きに変更し，依存関係による計算順序に基づいて規則 (3) と (4) を入れ替えることによって，下向き安全性を検査できる．式 e の節 n における下向き安全性の検査として用いるクエリは $antqp(e, n)$ で表す．

最終的に，クエリの伝播の結果，*isReal* が *true* であるならば，e を式の挿入の際に定義した新しい一時変数で置き換える．

5. 質問伝播に基づく大域ロード命令集約

本節で，PREQP を拡張した，**質問伝播に基づいて GLIA**（GLIA Based on Query Propagation，以降 GLIAQP と呼ぶ）を実現する手法を説明する．

GLIA は，ロード命令の出現ごとに，*Comp* や透過性といった節内の性質を表す局所的な性質を基に，$UpSafe$, $DownSafe$, *Earliest*, *Delayed*, *Latest*, *Insert* といった大域的な性質を表すデータフロー方程式の解を求めるためにプログラム全体を繰り返し解析しなければいけないので，解析コストが高くなる傾向があった．本手法は，PREQP を拡張することによって，プログラムの解析範囲を限定する．

本手法も PREQP と同様に，クエリが伝播された節にクエリの式が含まれるとき解 *true* を返す．しかしながら，クエリの解として *false* を返す際には，節内のロード命令のアクセス先を検査するように拡張する．同じ配列へアクセスするロード命令の出現以降，異なる配列へアクセスするロード命令が出現するとき，異なる配列へアクセスするロード命令の直前に新しく式を挿入することで，GLIA を実現する．

本手法では，クエリの解を，PREQP におけるクエリの解 *isAvail* に，同じ配列へアクセスするロード命令の出現を意味する $upSafe$ を加えた組み ($isAvail$, $upSafe$) によって表す．節内にクエリのロード命令と同じ配列にアクセスするロード命令が含まれているとき，$upSafe$ は *true* になる．

クエリの解を返すための規則を次のように定め，上から優先して適用する．

(1) 節 n における *isAvail* として *true* が得られたとき，(*true*, *true*) を返す．
(2) 節 n における *isAvail* として *false* が得られ，n にクエリの式と同じ配列にアクセスする配列参照が含まれているならば，(*false*, *true*) を返す．
(3) 節 n における *isAvail* として *false* が得られ，先行節の解の中で $upSafe$ が *true* のものが存在し，n に異なる配列にアクセスする配列参照が含まれているならば，n の入り口に新しく式を挿入し，(*true*, *true*) を返す．

上記の規則について PREQP を拡張した本手法の冗長性検査をアルゴリズムに示す．節 n の式 e について冗長性を調べるとき，クエリを伝播する関数 *propagate* と，節内を検査する関数 *local* を相互に再帰呼び出す．

関数 *local* は，クエリの伝播規則に対応している．27, 29, 35, 39, 42 行目は，それぞれ規則 (1), (2), (3), (4), (5) に対応している．27 行目で，すでに伝播された節の解

が得られている場合，その解を返す．34 行目の述語 $isSelf$ は，訪問した節がクエリを生成した節であることを意味する．

local 関数の返戻値は，$isAvail$, $upSafe$ に加えて，$isReal$ の 3 つ組み $(isAvail, isReal, upSafe)$ である．

関数 *propagate* は，各先行節の解を求めた後，どの先行節に新しく式を挿入するのかを決定する．10 行目に示すように，$isAvail$ が $false$ である先行節を式の挿入節として記録する．しかしながら，先行節に $isReal$ が $false$ であるものが存在するならばキャンセルする（17 行目）．13，21，24 行目は，それぞれ解を返すための規則 (1)，(2)，(3) に対応する．$isAvail$ が $true$ であれば，ただちに $(isAvail, upSafe)$ の解 $(true, true)$ を返す．一方，$isAvail$ が $false$ であるとき，節内にロード命令が含まれるならば（18 行目），ロード命令に含まれる配列参照のアクセス先アドレスの解析を行う．もし節内に含まれる配列参照が e と同じ配列ならば $(false, true)$ を返す（20 行目）．一方，e と異なる配列参照を含み，$upSafe$ が $true$ である先行節が存在するならば（22 行目），n に新しく式を挿入する（23 行目）．

アルゴリズム（GLIAQP の冗長性検査）
input: CFG
output: 配列 $query, answer, insertCand$

```
1:  Function propagate(e, n)
2:    let isDownSafe := antqp(e, n)
3:    and isReal := true
4:    foreach p ∈ pred(n)
5:      let (isAvail_p, isReal_p, upSafe_p) :=
          local(e, p)
6:      if (isAvail_p)
7:        if (¬isReal_p)
8:          isReal := false
9:      else
10:       insertCand[p] := e
11:   if (∏_{p∈pred(n)} isAvail_p ∨
12:     isReal ∧ isDownSafe)
13:     return (true, isReal, true)
14:   else
15:     foreach p ∈ pred(n)
16:       if (¬isAvail_p)
17:         insertCand[p] :=⊥
18:     if (isLoad(n))
19:       let pUpSafe := ∑_{p∈pred(n)} upSafe_p
20:       if (sameAddr(e, n))
21:         return (false, false, true)
22:       else if (pUpSafe)
23:         insertCand[n] := e
24:         return (true, true, true)
25:     return (false, false, false)

26: Function local(e, n)
27:   if (n = s)
28:     return (false, false, false)
29:   if (query[n] = e)
30:     if (answer[n] ≠⊥) return answer[n]
31:     else return (true, false, false)
32:   query[n] := e
33:   let rlt :=⊥ and
34:     (isAvail, isSelf) := isComp(e, n)
35:   if (isAvail)
36:     if (¬isSelf)
37:       insertCand[n] := e
38:     rlt := (true, true, true)
39:   else if (isMod(e, n))
40:     rlt := (false, false, false)
41:   else
42:     rlt := propagate(e, n)
43:   answer[n] := rlt
44:   return rlt
```

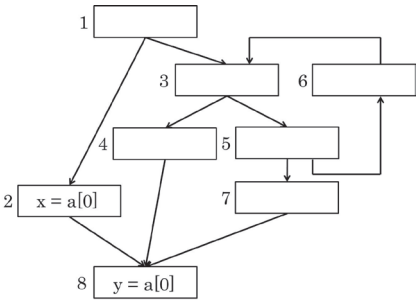


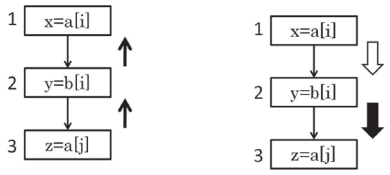
図 6 一貫性のない解
Fig. 6 Inconsistency answers

7 行目は，第 4 節で述べた不要なコード移動を防ぐための検査を行う．さらに，$isReal$ が $true$ であるかどうかの検査は，解の一貫性を保証することに貢献する [17]．GLIAQP は，$Comp$ が $true$ である節と，$isAvail$ が $false$ である節に新しく式を挿入するので，PREQP と同様に，どの節に式を挿入したのかが重要である．もし解の一貫性が損なわれると，本来は式の出現が得られなかった節において，$false$ であるべき解が，$true$ を得てしまい，正しいプログラム変形が行えない．

例：解の一貫性が保たれない例を図 6 に示す．図 6 は，節 8 の式 $a[0]$ に対して質問伝播を行い，節 2 と節 4 からクエリの解が得られた状況を示している．ここで，クエリの解を，$isAvail$ と $isReal$ に着目するために，考慮しない要素 $upSafe$ を _ で表現する．

節 2 と節 4 で得られる解は，それぞれ $(true, true, \_)$，$(false, false, \_)$ である．節 4 へ伝播したクエリに着目すると，節 3 から，節 6 へ伝播されたクエリは，節 5，3 と伝播される．質問伝播の規則 (2) によって，節 3 と節 5 における解は $(true, false, \_)$ である．したがって，節 8 から節 7，5 へと伝播したクエリは，節 5 の解として $(true, false, \_)$ を得るので，節 7 から得られる解は $(true, false, \_)$ である．

冗長性検査の結果，節 4 へ式 $a[0]$ を挿入することになるが，節 7 から節 8 へ到達する経路上に利用可能式が存在しないので，このプログラム変形は正しくない．これは，節



(a) クエリ伝播 Query propagation. (b) 解の伝播 Answer propagating.

図 7 質問伝播を用いた解析
Fig. 7 Analysis based on query propagation

3 の最終的な解 $(false, false, \_)$ が，節 5 から節 3 に伝播したクエリの解 $(true, false, \_)$ と異なり，一貫性が損なわれたからである．

解の一貫性を保つために，$isAvail$ が $true$ の解を得られたとき，$isReal$ を検査することによって，一貫性が損なわれないことを保証する．∎

冗長性検査の結果を用いて，$insertCand$ に記録されている節に式を挿入し，プログラム変形を行う．

例 : 図 7(a) は，図 1(a) の節 3 の配列参照 `a[j]` に GLIAQP を適用した際のクエリ伝播の様子を表す．節 1，および節 2 は配列参照 `a[j]` を含まないので，節 1 における $isAvail$ は $false$ である．したがって，$isAvail$ が $false$ なので，節 1 に含まれる配列参照のアクセス先アドレスを検査する．ここで，クエリの解を，$isAvail$ と $upSafe$ に着目するために，考慮しない要素 $isReal$ を _ で表現する．

節 1 の配列参照 `a[i]` は，クエリの配列と同じなので，節 1 の解は $(false, \_, true)$ である．また，節 2 は，先行節から解 $(false, \_, true)$ が得られ，節 2 の $isAvail$ は $false$ となる．したがって，節内の配列参照のアクセス先アドレスを検査する．節 2 は，クエリの配列とは異なる配列 `b` へアクセスする配列参照 `b[i]` が含まれ，$upSafe$ が $true$ である先行節が存在するので，節 2 の入口の新しく式を挿入し，解 $(true, \_, true)$ を返す．結果として，図 1(b) のプログラムが得られる．∎

6. 実験

本手法の効果を示すために，本手法を**並列コンパイラ向け共通インフラストラクチャ**（a COmpiler INfrastructure project）[3] 上で，低水準中間表現（Low-level Intermediate Representation）変換器として実現し，次の比較を行った．

GLIA： GLIA を適用する．

GLIAQP： GLIAQP（本手法）を適用する．

評価に使用したマシンのシステムパラメータを表 1 に示す．評価に使用したプログラムは，SPEC2000 ベンチマークのうち，CFP2000 の 3 つのプログラム（equake, art, ammp）と，CINT2000 の 7 つのプログラム（mcf, bzip2, gzip, gap, vpr, parser, twolf）である．

表 1 システムパラメータ
Table 1 System parameters

CPU	Intel Core i5-2320 3.00GHz
L1D Cache Memory	
Total size	32 KB
Line size	64 bytes
Number of Lines	512
Associativity	8
L2 Cache Memory	
Total size	256 KB
Line size	64 bytes
Number of Lines	4096
Associativity	8
L3 Cache Memory	
Total size	6144 KB
Line size	64 bytes
Number of Lines	98304
Associativity	12
OS	Ubuntu 12.04LTS

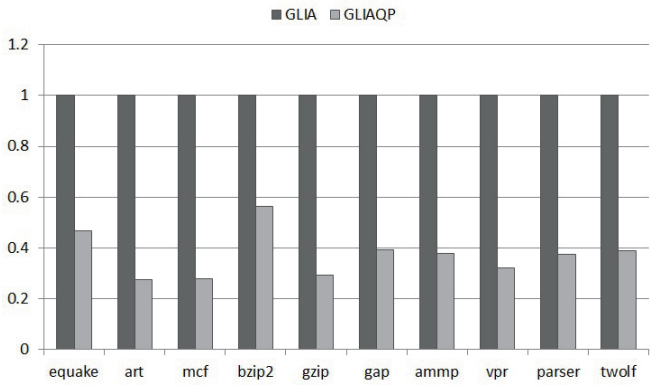


図 8 解析効率
Fig. 8 Analysis efficiency

各組合わせを適用した結果を**図 8** に示す．本手法を適用した結果，すべてのプログラムで解析効率が向上し，平均して約 62.7%の向上が得られた．特に，art において約 72.6%の向上が得られた．本手法は，質問伝播を用いて GLIA を実現しているので，実行効率の関しては変化が見られなかった．

上記の結果から，GLIA の実現には質問伝播が有効であることが示された．

7. 今後の課題

本手法は，クエリの解として $false$ が得られた際に，ロード命令のアクセス先アドレスを解析し，メモリアクセス順序を連続させることによって GLIA を実現している．

今後の課題としては，クエリ伝播の際に，ロード命令のアクセス先解析も行い，冗長性の除去とメモリアクセス順序を考慮したコード移動を同時に行うように拡張することが考えられる．この拡張によって，GLIA では除去される

冗長なロード命令が，キャッシュヒットを生じるように，同じ配列へアクセスするロード命令の近くに移動される．このようなコード移動によって，新しく挿入された変数の生存期間が短くなり，レジスタ圧力を低減させることが期待できる．ある変数 x に関してレジスタスピルが生じると，x の定義の直後にストア命令，使用の直前でロード命令が挿入される．したがって，GLIA を適用したことによって連続して行われていたメモリアクセス順序が崩れてしまう可能性がある．また，GLIA は，実行効率を低減させる投機的なコード移動を行う可能性がある [13]．生存期間が短くなると，投機的な移動を防ぐ場合も存在するので，実行効率が向上する可能性がある．

8. まとめ

本稿では，質問伝播と呼ばれる要求駆動型解析法に基づいて，ロード命令のメモリアクセス順序を連続させる大域ロード命令集約を実現する手法を提案した．

本手法は，クエリを伝播した結果，$false$ が得られたとき，$false$ を返す前に，各ロード命令のアクセス先アドレスを解析し，同じ配列へアクセスするロード命令の出現以降，異なる配列へアクセスするロード命令が出現した際に，新しく式を挿入することによって，メモリアクセス順序が連続されるようにプログラムを変形する．

本手法の効果を確かめるために，いくつかのベンチマークプログラムに本手法を適用したところ，データフロー解析に基づいた先行研究の手法と同じ実行効率を得ながら，解析効率が平均して約 63%向上することを確認した．

謝辞 本研究の一部は，科研（22300007）および科研（22500034）の助成を受けたものである．

参考文献

[1] Bodik, R., Gupta, R. and L., S. M.: Complete removal of redundant expressions, *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, New York, NY, USA, ACM, pp. 1–14 (1998).

[2] Cai, Q. and Xue, J.: Optimal and efficient speculation-based partial redundancy elimination, *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, Washington, DC, USA, IEEE Computer Society, pp. 91–102 (2003).

[3] COINS: http://coins-compiler.sourceforge.jp/.

[4] Dhamdhere, D. M.: A fast algorithm for code movement optimisation, *SIGPLAN Not.*, Vol. 23, No. 10, pp. 172–180 (1988).

[5] Dhamdhere, D. M.: E-path_ PRE:partial redundancy elimination made easy, *SIGPLAN Not.*, Vol. 37, No. 8, pp. 53–65 (2002).

[6] Dhamdhere, D. M. and Patil, H.: An elimination algorithm for bidirectional data flow problems using edge placement, *ACM Trans. Program. Lang. Syst.*, Vol. 15, No. 2, pp. 312–336 (1993).

[7] Kennedy, R., Chan, S., Liu, S.-M., Lo, R., Tu, P. and Chow, F.: Partial redundancy elimination in SSA form, *ACM Trans. Program. Lang. Syst.*, Vol. 21, No. 3, pp. 627–676 (1999).

[8] Knoop, J., Ruthing, O. and Steffen, B.: Lazy code motion, *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, New York, NY, USA, ACM, pp. 224–234 (1992).

[9] Knoop, J., Ruthing, O. and Steffen, B.: Optimal code motion: theory and practice, *ACM Trans. Program. Lang. Syst.*, Vol. 16, No. 4, pp. 1117–1155 (1994).

[10] Knoop, J., Ruthing, O. and Steffen, B.: Partial dead code elimination, *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, New York, NY, USA, ACM, pp. 147–158 (1994).

[11] Morel, E. and Renvoise, C.: Global optimization by suppression of partial redundancies, *Commun. ACM*, Vol. 22, No. 2, pp. 96–103 (1979).

[12] Rosen, B. K., Wegman, M. N. and Zadeck, F. K.: Global value numbers and redundant computations, *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, New York, NY, USA, ACM, pp. 12–27 (1988).

[13] Sumikawa, Y. and Takimoto, M.: Global Load Instruction Aggregation Based on Code Motion, *Proceedings of the 2012 5th International Symposium on Parallel Architectures, Algorithms and Programming*, PAAP '12, IEEE Computer Society (2012). , to be published.

[14] VanDrunen, T. and Hosking, A. L.: Value-based partial redundancy elimination, *In CC*, pp. 167–184 (2004).

[15] Zhou, H., Chen, W. and Chow, F.: An SSA-based algorithm for optimal speculative code motion under an execution profile, *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, New York, NY, USA, ACM, pp. 98–108 (2011).

[16] 今橋孝典， 伊藤陽, 佐々政孝：静的単一代入形式上で通常形式部分冗長除去を実現する汎用的手法，情報処理学会論文誌プログラミング（PRO），Vol. 49, No. SIG1(PRO35), pp. 84–95 (2008).

[17] 滝本宗宏：質問伝播に基づく投機的部分冗長除去，情報処理学会論文誌プログラミング（PRO），Vol. 2, No. 5, pp. 15–27 (2009).

[18] 中田育男：コンパイラの構成と最適化.