C-13

レジスタの有効利用を考慮した部分冗長性除去
Partial Redundancy Elimination Effectively Using Registers

佐藤 淳† 古関 聰‡ 小松 秀昭‡ 深澤 良彰†
Jun Sato Akira Koseki Hideaki Komatsu Yoshiaki Fukazawa

1. はじめに

部分冗長性除去（Partial Redundancy Elimination (PRE)）は，一度計算した値を保持しておき，それを再利用することで同じ計算を避ける手法である．一方で部分冗長性除去は置き換えた計算値の保持によりレジスタプレッシャを増加させる傾向がある．既にレジスタプレッシャの増加を最小にする手法 [1] が提案されているが，それでも解決できない余分なスピルコードの発生を避けるためには，部分冗長性除去の適用時に必要なレジスタ数が利用可能なレジスタ数を超えないことを保証する手法が必要となる．本手法では，レジスタリソースを付加的な情報としてデータフロー解析を行い，各アーキテクチャにあわせた効率的なレジスタの利用と最適な部分冗長性除去を実現する．

2. 本手法の特徴

部分冗長性除去の手法として Busy Code Motion や Lazy Code Motion[1] 等が挙げられる．計算を移動可能な最も早い時点に移動する Busy Code Motion が計算量最少にする手法であるが，置き換えた計算値の生存区間を長くするのでレジスタを多く使うことになりレジスタプレッシャが高まるという問題点があった．そこで，計算量をそのままで移動可能な最も遅い時点に移動する Lazy Code Motion が提案され，計算量最少でかつレジスタ生存区間最短になる方法であるとされている．

しかし，Lazy Code Motion を適用した場合でも，置き換えた計算値の生存区間内で必要なレジスタ数が利用可能なレジスタ数を超えることがある．レジスタプレッシャの増加によるスピルコードの発生は，冗長性除去から得る利益を相殺する引き金となる．冗長性除去の前後で，必要なレジスタ数が利用可能なレジスタ数を超える基本ブロックが増加することは，何らかのスピルコードを発生させる基本ブロックが増加するということと同義である．実際，Lazy Code Motion を適用するとこうした基本ブロックが 5-10%増加すると言われている [2]．

Busy Code Motion や Lazy Code Motion は実際のレジスタ数を考慮したアルゴリズムではないので，利用可能なレジスタ数が十分でない場合，実際に生成されたコード中にレジスタが十分に有効利用されない部分が発生していた．

このようなレジスタプレッシャの問題に取り組んだ手法 [2] では，レジスタリソースを付加的な情報として利用して，冗長性除去とレジスタプレッシャのバランスを考慮したアルゴリズムを展開している．しかし，この手法では置き換えた計算値をレジスタに格納するケースしか考えておらず，利用可能なレジスタ数で部分冗長性除去を適用できなければ冗長性除去の機会を失うことになる．

本手法では，利用可能なレジスタ数を超える場合でも，置き換えた計算値をメモリに格納するケースや，冗長性除去を適用しないケースのことまでも考慮したアルゴリズムを組む．そして，部分冗長性除去の適用過程で，メモリオペランドを持つ命令や実行にかかるコストの低い命令をより積極的に利用する．このような命令を効率よく利用することで，各アーキテクチャにあわせた効率的なレジスタの利用と最適な部分冗長性除去を実現する．

†早稲田大学理工学部
‡日本 IBM(株) 東京基礎研究所

3. 本手法の概要

始めに本手法の概要を提示し，それからアルゴリズムの重要なステップに関して詳細に説明する．

本手法では，Lazy Code Motion を適用できる計算に対して，次の 3 種類の適用パターンに分類する．Lazy Code Motion の適用可能な計算を求めるアルゴリズムに関しては文献 [1] に詳細に書かれている．

1. Lazy Code Motion を適用し，置き換えた計算値をレジスタに格納 (3.1 節)
2. Busy Code Motion を適用し，置き換えた計算値をメモリに格納 (3.2 節)
3. 部分冗長性除去を行わず，対象の計算は再計算 (Rematerialization)(3.3 節)

適用パターンの分類を行うのに，メモリオペランドを持つ命令や実行にかかるコストの低い命令を考慮する．置き換えた計算値をメモリに格納する場合，メモリオペランドを持つ命令を利用することで効果的な部分冗長性除去を実現する．また，実行にかかるコストの低い命令に対しては，部分冗長性除去を適用せず再計算する．レジスタリソースを付加的な情報として利用する手法としてこのような命令を効率よく利用することにより，限られたレジスタ内で各アーキテクチャにあわせた効率的なレジスタの利用と最適な部分冗長性除去を実現する．

3.1 Lazy Code Motion on Registers

このパターンでは，Lazy Code Motion を適用し，置き換えた計算値をレジスタに格納する．置き換えた計算値の生存区間内で必要なレジスタ数が利用可能なレジスタ数を超えない場合，すなわち，レジスタに余裕がある場合に本パターンを適用する．レジスタに余裕がある場合は，スピルコードが発生しないという前提上，このパターンが有効である．Lazy Code Motion の適用により，できるだけ計算を遅らせる．

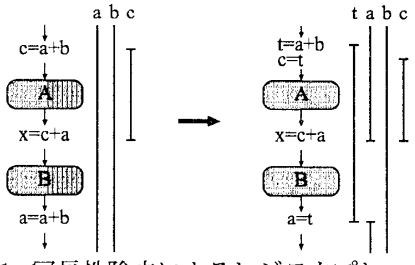

図 1: 冗長性除去によるレジスタプレッシャ

必要なレジスタ数の計算について，冗長性除去の性質の観点から説明しておく．図 1 では，計算 (a+b) を置き換えた値 (t) を保持しておくことにより冗長性除去を行っ

ている．ブロックAでは置き換えた計算値(t)により必要なレジスタ数が1つ増加する．しかし，ブロックBでは置き換えた計算値(t)により必要なレジスタ数は1つ増加するが，変数(a)の生存区間が取り除かれることで必要なレジスタ数が1つ減少し，結局レジスタプレッシャは変化しない．このように，計算が置き換えられることでレジスタプレッシャを減らす場合もあり，冗長性除去によって一概にレジスタプレッシャが高まるとは言えず，ブロックごとに必要なレジスタ数を分析する必要がある．

3.2 Busy Code Motion on Memory

置き換えた計算値によるレジスタプレッシャの増加でスピルコードを発生させてしまっては意味がない．このパターンでは，Busy Code Motionを適用し，置き換えた計算値をメモリに格納する．置き換えた計算値の生存区間内で必要なレジスタ数が利用可能なレジスタ数を超える場合，すなわち，レジスタに余裕がない場合に本パターンを適用する．置き換えた計算値をメモリオペランドで参照可能なら，レジスタプレッシャが高い場合でも必要なレジスタ数を増やさずに効果的な部分冗長性除去を行なうことができる．

3.3 No Code Motion

このパターンでは，コードの移動を行わない．すなわち，部分冗長性除去を行わないため対象の計算は再計算されることになる．実行にかかるコストの低い命令は，置き換えた計算値を保持してメモリへのストア／ロードをするのでなく，必要なときに再計算した方が良い．

3.4 適用パターンの分類手順

以下に，適用パターンを分類する手順を示す．以下の手順をLazy Code Motionを適用可能である計算がなくなるまで繰り返す．複数ある場合，実行にかかるコストの高い計算から順に適用する．

(1) Lazy Code Motionが適用可能な計算のうち，実行にかかるコストの最も高い計算を選ぶ．

(2) 対象の計算に対してLazy Code Motionを適用した際，置き換えた計算値の生存区間内で必要なレジスタ数が利用可能なレジスタ数を超えない場合のみ，Lazy Code Motionを適用して，置き換えた計算値をレジスタに格納して，(1)へ戻る．

(3) 対象の計算に対してBuzy Code Motionを適用した際，置き換えた計算値をメモリに格納しメモリオペランドで参照することができる場合のみ，Busy Code Motionを適用して，置き換えた計算値をメモリに格納して，(1)へ戻る．

(4) 対象の計算に対してBusy Code Motionを適用した際，置き換えた計算値を保持することでかかるストア／ロード命令の時間よりも，保持せず再計算した時間の方が速い場合(a)へ．そうでない場合(b)へ．

- **(a)** 冗長性除去を行わず再計算する．(1)へ戻る．
- **(b)** Busy Code Motionを適用して，置き換えた計算値をメモリに格納して，(1)へ戻る．

4. 本手法の適用例

本手法の適用例を図2に示す．図2(a)の配列演算を含むプログラムに対して手法[1],[2], 本手法を適用した例が図2(b),(c),(d)である．

従来の手法と比較して以下のような特徴があげられる．

1. スピルコードの減少

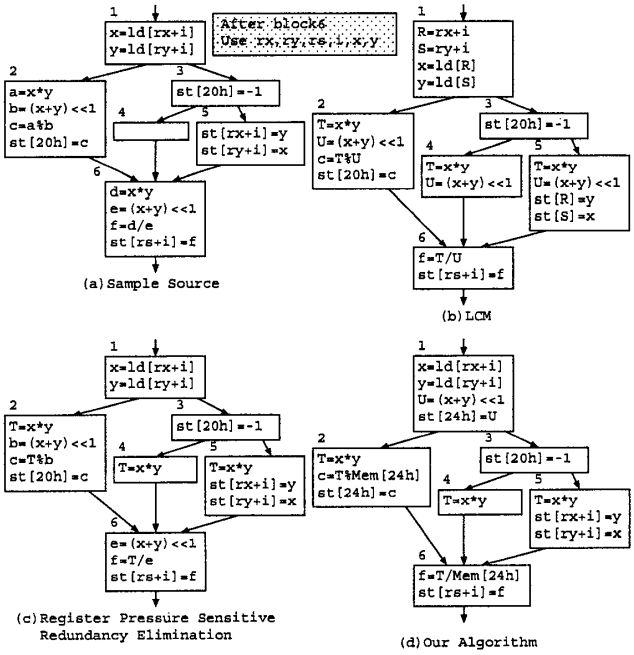

図2: 本手法適用例

2. メモリオペランドを持つ命令の有効利用
3. 再計算した方が速い場合の部分冗長性除去の不適用
4. 各アーキテクチャ向けのレジスタ有効利用

1.に関しては，手法[1]と違い，レジスタプレッシャを意識したことでレジスタプレッシャを減少させたことによる．図2(b)のブロック2,4で発生していたスピルコードは図2(d)では発生しない．2.に関しては，冗長性除去で置き換えた計算値をストアしてその値をメモリオペランドで参照することで，レジスタプレッシャが高いときでもメモリを使って効率よく冗長性除去が適用できた．図2(d)での，(x + y)<<1の計算がそうである．3.に関しては，計算のコストを考慮することで，再計算が簡単にできる命令の対処を効率化できた．図2(d)では，rx+i, ry+iの計算はストア／ロード命令のアドレス計算なのでメモリオペランドで参照ができないが，置き換えた計算値をメモリに保持することでかかるストア／ロード命令の時間よりも，保持せず再計算した時間の方が速いと判断され，冗長性除去を行わず再計算している．4.に関しては，アーキテクチャごとにレジスタ数の設定を変化させることで本手法が対応できる性質による．

5. おわりに

本論文で単にレジスタプレッシャの増加を最小にしようとする現在の部分冗長性除去の手法が不適当であることを示した．このような手法ではスピルコードを発生させる．このために，レジスタリソースを付加的な情報として用意し、利用可能レジスタ数の制約の中で部分冗長性除去を実行する手法を提案した．

参考文献

[1] J.Knoop, O.Runthing, B.Steffen: "Lazy Code Motion", Proc. of PLDI, Vol.27, No.7, 1992, pp.224-234

[2] R.Gupta, R.Bodik: "Register Pressure Sensitive Redundancy Elimination", Proc. of Inter. Conf. on Compiler Construction, 1999, pp.107-121