コンパイラによるロード・ストア負荷の軽減

服部 直也　飯塚 大介　坂井 修一　田中 英彦
東京大学工学系研究科

概要

プログラム中に内在するメモリアクセス命令は、プロセッサ処理のボトルネックになる可能性がある。そのためコンパイラは Register Promotion という最適化を行ってメモリ変数をレジスタ変数に格上げし、Register Allocation と合わせてメモリアクセス数を削減している。しかし、通常の Register Promotion は関数内のみに限定されて行われており、メモリアクセスを十分に削減しているとは言い切れない。本稿では引数レジスタ・返値レジスタを用いて、Register Promotion を関数間で行う手法を提案する。またメモリアクセス削減の初期評価として、レジスタ無限大モデルにおける性能を示し、関数内 Register Promotion と比べて性能が向上することを確認した。

Compiler Support for Load / Store Pressure

Naoya Hattori　Daisuke Iizuka　Shuichi Sakai　Hidehiko Tanaka
Graduate School of Engineering, University of Tokyo

Abstract

Memory access instructions are potentially bottlenecks on processor performance. To reduce them, both Register Allocation and Register Promotion are performed in compiling phase. Register Promotion is an important function to reduce explicit memory access instructions before Register Allocation. Many algorithms have been presented for Register Promotion, but all of them are only intraprocedural. In this paper, we propose Interprocedural algorithm to overcome intraprocedural limitation and demonstrate its capcacity in removing memory access.

1　はじめに

近年プロセッサの持つ並列処理能力が向上しているが、ALU や FPU 等と異なりロード・ストアユニットの処理並列度はメモリアクセス命令間の曖昧な依存のために向上させ難い。また、今後プロセッサの並列度が上がるにつれ、この傾向は顕著になると考えられる。一方で、プログラム中にはメモリアクセス命令が少なからず含まれており、プロセッサ処理のボトルネックになると考えられる。この問題に対処するため、ハードウェアだけでなくコンパイラの分野においてもキャッシュの活用、レイテンシの隠蔽など様々な最適化の研究がなされてきたが、この中でメモリアクセス回数を削減する Register Promotion (RP) という最適化も注目されている。

通常コンパイラは Register Allocation を行って、プログラム中の Scalor(非配列) 変数を可能な限りレジスタ変数にする。しかし、一部の Scalor 変数は Register Allocation の候補にすら入っていないため、仮にレジスタが無限に利用できたとしてもメモリアクセスが少なからず残る。近年、メモリアクセスを減らす為に論理レジスタを 64 あるいは 128 個備えたようなプロセッサも現れているが、これでは拡張したレジスタが有効に活用できない。

Register Allocation の候補から外れているのは、(1) メモリアドレスを使用される変数
(2) Global 変数 の 2 種であり、両者ともアクセスされる位置が明確でないために、完全なレジスタ変数にはできない。しかしこれらの変数もアクセス位置が明確な範囲ではレジスタ変数に格上げ (Promotion) することができる。この最適化が RP である。

従来の RP は関数内のみで行われるため、関数

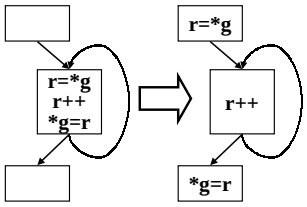

図 1: 単純なループに対する RP

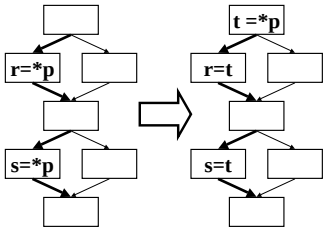

図 2: Speculative RP
(Profile データが必要)

間に跨ってアクセスされるメモリ変数をレジスタ変数に Promotion できなかった。本稿では RP を関数間に拡張し、これらのメモリ変数を 引数レジスタ・返値レジスタに Promotion する手法を提案・評価する。

2 Register Promotion

RP[2, 5, 4] はメモリ変数を一時的にレジスタ変数に格上げする最適化である。この格上げによって Register Allocation の候補となる変数が増え、代わりに明示的なメモリアクセスが削減される。また RP は、本来必要のないメモリアクセスを削減するという意味で Partial Redundancy Elimination(PRE) [1, 6, 3, 7] に似た部分も多い。

例えば図 1左はループの中でグローバル変数 $(*g)$ をインクリメントするようなプログラムを示している。ここで、四角で囲まれた部分は Basic Block を、矢印は分岐を表している。最適化前のコードは左のようにループ内にメモリアクセスを含んでしまうため、高速動作は難しい。しかしこの例ではグローバル変数 $(*g)$ のアクセス位置は明確であるため、RP を適用することで図 1右のように変形される。また、図 2左のような部分プログラムでは、どのように Load を配置しても、Load の実行回数が増えてしまう可能性がある。しかし Profile 情報等により太い矢印の分岐回数が多いことがわかれば、投機的に Load を配置することにより、アクセス命令の実行回数を削減することができる (図 2右)。

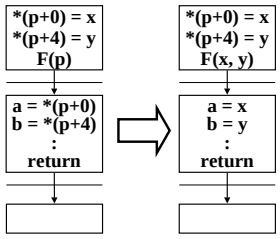

図 3: ポインタ引数を介したメモリアクセス (左) と関数間 RP で最適化されたコード (右)

従来の RP では、関数間のポインタ解析を行うことはあっても、コード変形自体は関数内のみで行っていた。そのため (1) Global 変数を介した関数間の冗長アクセス (2) ポインタ引数を介した関数間の冗長アクセスを Promotion することができない。例えば後者は、構造体へのポインタを関数間で受け渡す際に多発するが、RP を関数間で行えばこれらのアクセスもレジスタ上で行わせることができる (図 3)。

3 提案する関数間 RP

本手法は以下の手順で行う。

1. 関数間ポインタ解析を行う。
2. 関数毎に 引数・返値の追加に対応した RP を適用する
3. 変化がなくなるまで 手続き 2 を繰り返す
4. 使用していない引数・返値を削除する

RP はポインタ解析を必要とするが、ポインタ解析部分は本稿の範囲から外れるので割愛する。ここでは、単に関数間ポインタ解析のデータが利用できるものとする。

本手法は解析範囲の 2 乗オーダーの計算を行うため、一度にプログラム全体を最適化しようとすると計算量が爆発する。そこで一回の最適化範囲を関数単位に限定し、関数毎の RP を収束するまで繰り返すことで行うことで全体を最適化する。ただし関数間最適化に必要な情報は、次に説明する仮想アクセス命令により、関数間で伝搬される。

メモリアクセス命令としては通常の Load, Store の他に、2 つの仮想アクセス命令 Pre-Loaded, Post-Stored を考える (表 1)。 Pre-Loaded は $*p$

命令	略記法
Load	$x \leftarrow *p$
Store	$*p \leftarrow x$
Pre-Loaded	$x == *p$
Post-Stored	$(*p)$

表 1: (仮想) アクセス命令と図中の略記法

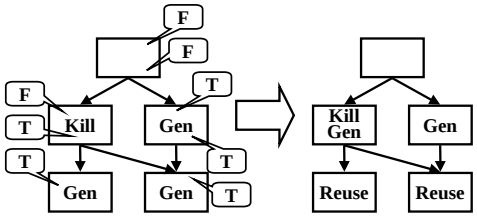

図 4: DownSafety の解析例と PRE

の値は既に他の関数で Promotion されていてその値が x であることを示す。Post-Stored は、他の関数が Store するためにそこで Store する必要がないことを意味する。

3.1 仮想命令に対応した 関数毎の RP

本手法には、前節で述べた仮想命令に対応した関数毎の RP が必要になる。ベースとなる関数内 RP には任意のアルゴリズムが適用できるが、ここでは今回実装した Busy Code Motion をベースにしたアルゴリズムを説明する。このアルゴリズムは以下の手順で行う。

1. 未最適化アクセス命令を 1 つ選択
2. 選択された命令と関数内の命令の依存調査
3. Safety, Availability の計算
4. アクセス命令配置位置の決定
5. コード変形
6. 収束するまで手続き 1 〜 5 を繰り返す

3.1.1 命令間の依存調査

ポインタ解析データを元に、選択された命令と関数内の各命令間の依存を調べる。依存関係として、確実な同アドレスへのアクセス以外に、Load するかもしれない (May-Load) と Store するかもしれない (May-Store) を定義する。明らかに選択された命令と同アドレスをアクセスしない命令は無視する (nop として扱う)。また、union 等で生じる型が異なるアクセスに関しては、May-Load, May-Store と同等に扱い、RP には用いないものとする。

3.1.2 Safety, Availability の計算

コード変形の際には、以下の 2 つの条件を満たす必要がある。

(a) 本来起きないはずの例外を起こさない
(b) アクセス命令の実行回数が増えない

$$DownSafety\_out[exit] = False \quad (1)$$

$$DownSafety\_out[I] = \bigcap_{S \in succ[I]} DownSafety\_in[S] \quad (2)$$

$$DownSafety\_in[I] = Gen[I] \mid (!Kill[I] \ \&\ DownSafety\_out[I]) \quad (3)$$

図 5: DownSafety の求め方

これには、PRE の *DownSafety* (下に必ず同じ計算が現れるという属性) に相当する解析が必要になる。図 4は PRE の動作を示しており、*Gen* が選択された冗長な命令、*Kill* が *Gen* への入力を上書きする命令、*Reuse* は *Gen* の結果を再利用する命令 を表している。左のコードは PRE によって、右のコードに変換されるが、*DownSafety* な位置に *Gen* を配置しているため、新たな例外を起こしたり、*Gen* の実行回数が増えたりすることはありえない。ただし、*Gen* が例外を起こさないことがわかっている場合や、各点の実行回数がわかっている場合には *DownSafety* の解析は必要無い。

DownSafety は図 4のように各命令の上下の端で計算され、範囲内の全ての命令について図 5の (2), (3) 式を、変化が無くなるまで繰り返し計算することで解を得る。*Gen[I]*、*Kill* は 命令 I が *Gen*、*Kill* であるかを表し、*DownSafety_in[I]* は I の直前、*DownSafety_out[I]* は I の直後 における *DownSafety* を表す。式 (1) は、関数終端での *DownSafety_out* が常に *False* であることを示す。また、*succ[I]* は I の直後に実行される可能性のある命令の集合である。

詳細は次節で述べるが、RP では 表 2に示した

	DownSafety(PRE)	L-Safety	L-Avail	S-Safety	S-Avail
方向	↓	↓	↑	↑	↓
Gen	冗長計算	Load	Load, Pre-Loaded Store	Store	Store Post-Stored
Kill	Gen の結果を変更する命令	Store, May-Store	May-Store	May-Load May-Store	May-Load, May-Store

表 2: Safty, Availability の求め方

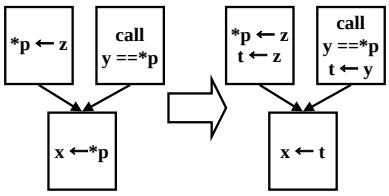

図 6: Avail の使用例

4 つの属性を計算する必要がある。このうち Load の DownSafety (L-Safety) と Store の UpSafety (S-Safety) は、Load / Store を配置する位置を特定するために用いる。また、Load の Availavility (L-Avail)、Store の Availability (S-Avail) は各位置に本当に Load / Store が必要か否かを判定するために用いる。それぞれは表 2に示した様に、DownSafety における Gen と Kill を差し替えることで求め、また表中で DownSafety と異なる方向の属性は上下を逆に計算することで求める。

3.1.3 アクセス命令の配置

Busy Code Motion では、命令を DownSafety である最上位点に配置する。図 4の場合は、Kill の直下と右上の Gen の直前が該当する。そして元あった Gen を Reuse に変換する。PRE の場合、Reuse としては代入命令が使用される。

RP に於いても同様の処理を行う。Load 削減の際は L-Safety である最上位点のうち、L-Avail でない位置に Load を挿入し、元の Load は全て代入命令 (Load の Reuse に相当) に置き換える (図 6)。

Store 削減の際は S-Safety である最下位点のうち、S-Avail でない位置に Store を挿入し、元の Store は全て削除 (Store の Reuse に相当) する。

3.2 引数・返値の追加

本手法では、関数間に跨ってアクセスされるメモリ変数を引数レジスタ・返値レジスタに Promotion させる。通常の Register Allocation では、引数と返値は固定数のレジスタを用いて受け渡される。レジスタが足りる限りレジスタが割り当てられるので、引数・返値も広い意味で Register Allocation の候補に含まれる。よって、メモリ変数を引数・返値に昇格することで関数間で RP を行うことができる。

尚、引数・返値への Promotion 候補は Caller / Callee の双方でアクセスできる必要があるため、(1) Global Scalor 変数。(2) アドレスが 引数 + Const で示されるメモリ変数。の 2 種に限定される。

3.2.1 引数の追加

メモリ変数を引数に Promotion する際も、関数内の RP と同様に条件 (a), (b) を満たす必要があるため、(1) Callee 側でそのメモリ変数を必ず Load する (関数の先頭で L-Safety) あるいは (2) 全ての CallSite で必ず L-Avail であるのいずれかを満たしている場合に限られる。

本手法では関数内の RP の際に、これらの条件を満たすメモリ変数に対して以下の 3 つの処理を行い、引数へ Promotion する (図 7左→中)。

1. 引数を追加する
2. CallSite の直前に Load を追加する
3. Callee に Pre-Loaded を追加する

CallSite / Callee に双方に命令を追加することで、残りの処理は関数内の RP に帰着する (図 7中→右)。

3.2.2 返値の追加

返値の追加処理も引数の場合と同様に、(1) Callee の return 命令が S-Safety である場合あるいは

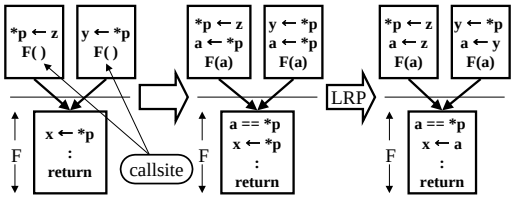

図 7: 引数の追加

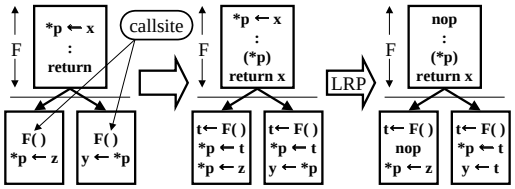

図 8: 返値の追加 (store の移動有り)

(2) 全ての CallSite で S-Avail である場合には以下の 3 つの処理を行い、Store を関数外へ出す (図 8左→中)。

1. 返値を追加する
2. CallSite の直後に Store を追加する
3. Callee の return 命令の直前に Post-Stored を追加する

この後、関数内の RP によりアクセスが削減される (図 8中→右)。

ただし、上記条件を満たさない場合でも (3) Callee の return 命令が L-Avail である場合あるいは (4) 全ての CallSite で必ず L-Avail である場合には Load を削減するために返値を追加する (図 9左→中)。この場合の処理は以下のようになる。

1. 返値を追加する
2. CallSite の直後に Pre-Loaded を追加する
3. Callee の return 命令の直前に Load を追加する

3.3 spill を増やさない heuristic

前節で述べた手法を用いると Global Scalor 変数を全てレジスタ変数に格上げすることができるが、代わりに大量の spill が発生する恐れがある。図 10 は compress95 に bigtest.in を入力した場合の load と spill-in の実行回数を示している。

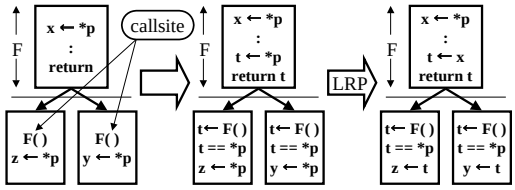

図 9: 返値の追加 (store の移動無し)

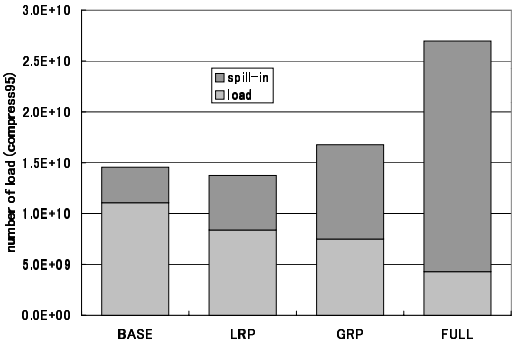

図 10: SPARC allocation rule での spill 数

この調査では Register Allocation Rule として SPARC モデル (引数レジスタ 6 本、返値レジスタ 1 本) を仮定した。ただし spill 数を正確に把握するために Register Window 命令の使用は避けた。項目はそれぞれ 未最適化コード (BASE)、関数内 RP 後のコード (LRP)、全ての Global Scalor アクセスを取り除いたコード (FULL) と後述するヒューリスティックを用いた関数間 RP 後のコード (GRP) である。FULL では spill 数が大量に増えている。

spill が増えたのは変数の Life Time が長くなったためであるが、これは主に call 命令を跨いで生きる変数が増加したためと考えられる。
そこで FULL に対し、call を跨いで生きる変数を加えない という制約を加える。この制約は 前節の アクセス命令と call の関係を全て May-Store とすることで満たすことができる。

制約を加えた場合の結果が図 10の GRP であるが、FULL と比較すると格段に spill 数が抑えられている。また、SPARC Allocation Rule では GRP が LRP よりも総 load 数が多くなっているが、これは Allocation Rule、特に返値レジスタ数を変更することで容易に改善されると考えている。

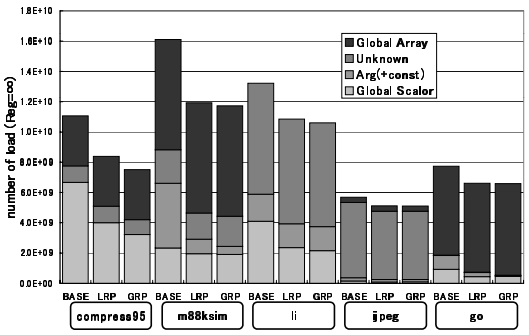

図 11: レジスタ∞モデルに置ける RP 性能比較

4 評価

提案手法を評価するために、富士通研究所の C コンパイラ [8] の中間コードに対し、本手法のロード削減部分を実装した。本研究は最終コードのメモリアクセス数を減少させることを目的としているが、現在、Register Allocation 部分までの研究に至っていないため、レジスタ無限大モデルで評価を行った。SPECint95 の 5 つのプログラムに関する結果を図 11に示す。各プログラムへの入力としては ref を使用している。縦軸は、レジスタ無限大モデルでのロードの数を示しており、項目は前節のものと同じである。また、グラフ中の Global Scalor は FULL アルゴリズムで全削除可能なアクセス、Arg + Const は、ポインタ引数を解したアクセスであり、この 2 つが GRP の対象になる。Global Array は RP の対象外であるため LRP、 GRP ともアクセス数は削減できない。

このグラフでは ijpeg と go は LRP と GRP の差が現れていない。これは GRP の対象となる Global Scalor や Arg + Const へのアクセスが少ないためであり、これらに対しては GRP を適用する価値は薄いと言える。しかし、compress95, m88ksim, li に関しては GRP の対象を多く含んでいるため、GRP は LRP より多くの Load を削減できている。

5 まとめと今後の課題

本稿では、メモリアクセス削減のための関数間解析の初期評価として、Register Promotion を関数間に拡張してレジスタ無限大モデルでのロード・ストア数を削減するアルゴリズムを提案した。また実際にロード削減の部分を実装し、spill を抑えるheuristic を加えた場合でも、LRP より多くのロードを削減できていることを確認した。

今後の課題としては、以下の点が挙げられる。今回用いたヒューリスティックは、実際のレジスタ数や callee の大きさを考慮していない。レジスタ有限モデルに適用させるために、より賢いヒューリスティックを考案する必要がある。また、Busy Code Motion ではなく、Lazy Code Motion ベースの RP も検討する必要がある。今回は SPARC モデルでの spill 数による評価であったが、任意のレジスタ数・ Allocation Rule で評価を行う必要がある。更に、Profile データを効果的に使用すれば 関数内・関数間の RP の性能が向上するため、Profile を用いた場合の性能を評価しなおす必要があると考えている。

謝辞

本研究を進めるにあたり、富士通研究所の小沢年弘氏、木村康則氏にコンパイラの使用許諾及び定期的な御助言を頂きました。深く感謝致します。

参考文献

[1] Bodik, R. and Gupta, R. and Soffa, M.L. Complete Removal of Redundant Expressions. *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pp. 1–14, 1998.

[2] Keith D. Cooper and John Lu. Register Promotion in C Programs. *PLDI*, pp. 308–319, 1997.

[3] Gupta, R. and Bodik, R. Register Pressure Sensitive Redundancy Elimination. *Lecture Notes in Computer Science*, No. 1575, pp. 107–121, 1999.

[4] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register Promotion by Sparse Partial Redundancy Elimination of Loads and Stores. *SIGPLAN*, pp. 26–37, 1998.

[5] A.V.S. Sastry and Roy D.C. Ju. A New Algorithm for Scalar Register Promotion Based on SSA Form. *SIGPLAN*, pp. 15–25, 1998.

[6] Steffen, B. Property Oriented Expansion. *Lecture Notes in Computer Science*, No. 1145, pp. 22–41, 1996.

[7] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.

[8] 飯塚 大介 他. C コンパイラにおけるループ最適化の検討. *HPC 77*, pp. 65–70, 1999.