計算機アーキテクチャ 144−13
（２００１．７．２６）

関数間最適化による冗長メモリアクセスの削減

服部 直也 † 峯 博史 ‡ 坂井 修一 † 田中 英彦 †
† 東京大学情報理工学系研究科 ‡ 東京大学工学系研究科

Abstract

メモリアクセス命令は ALU, FPU 命令などと比べて、低速であり、並列度も上げ難いという特徴がある。そのため、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っている。従来、最適化コンパイラは、関数内の Register Promotion を適用してメモリアクセス命令の実行回数を削減していたが、関数間にまたがるような冗長なメモリアクセスを削減することはできなかった。そこで我々は関数間 Register Promotion を提案している。本稿では、更に強化したアルゴリズムを提案し、関数間 Register Allocation と合わせた、メモリアクセス削減最適化全体の評価を行った。SPECint ベンチマークのいくつかのアプリケーションを用いて評価した結果、最大で 50% の Load と 26% の Store の削減に成功した。

Redundant Memory Access Elimination via Interprocedural Register Promotion/Allocation

Naoya Hattori † Hiroshi Mine ‡ Shuichi Sakai † Hidehiko Tanaka †
†Graduate School of Information Science and Technology, University of Tokyo
‡Graduate School of Engineering, University of Tokyo

Abstract

Memory access instructions are relatively difficult to execute fast, and to execute in parallel. They are the bottleneck of processor performance. To reduce the execution count of memory instructions, intraprocedural register promotion is performed in many optimizing compilers. But redundant memory access instructions across procedure boundary still remain. To eliminate such instructions, we have proposed “Interprocedural Register Promotion”. In this paper, we propose more powerful method. Combining with Interprocedural Register Allocation, we evaluate our optimization system that reduces memory access instructions. Our results on some applications in SPECint benchmark suites show up to 50% of Loads and 26% of Stores are eliminated.

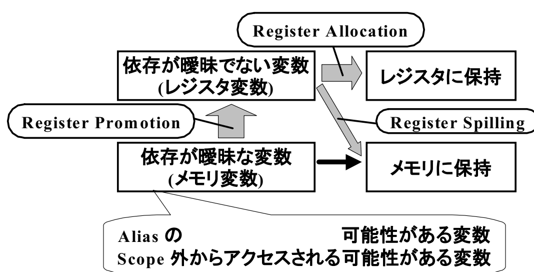

Figure 1: Register Promotion と Register Allocation

1 Introduction

近年プロセッサの備える並列処理能力や動作周波数は向上しており、命令の処理速度が向上している。しかし、メモリアクセス命令は ALU 命令や FPU 命令と比べて、高速動作が難しく、処理並列度も向上させ難いという特性があるため、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に、最適化コンパイラは Register Promotion と Register Allocation という直行する２種類の最適化を行っている (Figure1)。Register Allocation は依存が曖昧でない Scalar (非配列) 変数に Register を割り当てる最適化である。この際プロセッサの備えるレジスタ数が不足すれば Spill と呼ばれるメモリアクセスが生じるが、可能な限り変数はレジスタ上に保持される。これに対し、Register Promotion は依存が曖昧な変数を解析して、曖昧性を否定できる変数を Register Allocation の対象に加える最適化である。

これまで Register Promotion は関数内のみで行われていたため、関数間に跨ってアクセスされるメモリ変数は Promotion されなかった。そこで我々は Register Promotion 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 する手法を提案し、単体での

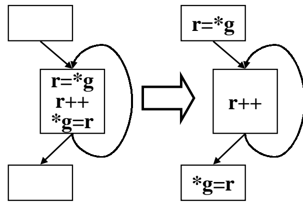

Figure 2: 単純なループに対する Register Promotion

Load 削減性能を評価した [8]。本稿では更にアルゴリズムを強化し、関数間 Register Allocation と組み合わせた総合的な評価を行う。

2 Register Promotion

Register Promotion [2, 5, 4] はメモリ変数をレジスタ変数に格上げする最適化である。結果として冗長なメモリアクセス命令が削減されるが、これは冗長演算を削減する最適化 Partial Redundancy Elimination [1, 6, 3, 7] を応用した最適化になっている。

Partial Redundancy Elimination では、

Safe :
同一内容の計算が下方に必ず存在する
(この位置で計算しても無駄にならず、
余計な例外を発生させない)。

Avail :
同一内容の計算が上方に必ず存在する
(この計算は完全に冗長)。

という 2 つの属性をプログラム中の各点で計算し、Safe な位置に計算を挿入し、Avail な位置の計算を「以前の計算結果を使い回すコード」に置き換えることで冗長性を除去する。

この Partial Redundancy Elimination をメモリアクセス命令に適用すればRegister Promotion になるのだが、メモリアクセス命令には曖昧な依存が存在するため、Safe や Avail の計算にポインタ解析が必要となる。

例えば Figure2 左はループの中でメモリ変数 $(*g)$ 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いるが、この範囲内ではメモリ変数 $(*g)$ へのアクセスに曖昧性がないと仮定すると、Register Promotion を適用することで Figure2 右のように変形することができる。

従来の Register Promotion 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 Global 変数を介した関数間の冗長アクセスやポインタ引数を介した関数間の冗長アクセスを Promotion することができなかった。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、Register Promotion 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 (Figure3)。

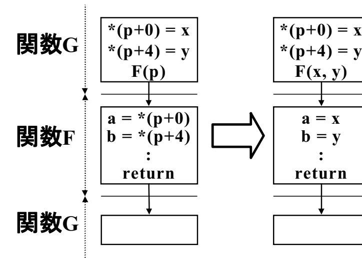

Figure 3: ポインタ引数を介したメモリアクセスと関数間 Register Promotion

3 Interface Register Promotion

本章では我々が提案する Interface Register Promotion のアルゴリズムを説明する。このアルゴリズムは文献 [8] で述べたアルゴリズムに Store の削減を加えたものである。

3.1 Overview of Approach

本手法では、関数間でメモリアクセスを伴わずに値を受け渡す方法として、引数や返値という関数間 Interface に着目する。通常、関数の引数と返値は Conventional Rule に定められたレジスタを用いて受け渡されるため、メモリ変数を引数・返値に Promotion することにより、関数間でメモリアクセスを削減することができる。

また、本手法では計算量の爆発を防ぐために一回の最適化範囲を関数単位に限定し、関数毎の Register Promotion を Leaf Function から順に繰り返すことで全体を最適化する。この際、他の関数の挙動を伝達するために、仮想アクセス命令 Pre-Loaded と Post-Stored を用いる。本稿における各命令の定義を Table1 に示す。Pre-Loaded x = *p は $*p$ 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 p は、他の関数が Store することが保証されているため、そこでアドレス p へ Store する必要がないことを意味する。Register Promotion の際には、適切な命令をその関数の呼び出し元 (CallSite) に追加することで必要な情報を伝搬させる。

また、本手法では引数・返値の変更を行うが、

命令	略記法	意味
Load	ld :$x = *p$	アドレス p から値を読み x へ書く
Store	st :$*p = x$	アドレス p に x の値を書く
Pre-Loaded	pld :$x = *p$	アドレス p の値は x に保持されている
Post-Stored	pst :$*p$	アドレス p は読まれる前に必ず書かれる
arg-Φ	a = arg-Φ(...)	仮引数と実引数の binding を表す
ret-Λ	ret-Λ(...) = a	仮返値と実返値の binding を表す

Table 1: 本稿で用いる命令の定義と略記法

	条件	追加 Interface	関数内に配置	CallSite に配置
(1)	関数先頭で Load Safe	arg-Φ	先頭に PreLoaded	直前に Load
(2)	関数先頭で Store Avail	なし	なし	直前に PostStored
(3)	関数末尾で Store Safe	ret-Λ	末尾に PostStored	直後に Store
(4)	関数末尾で Load Avail	ret-Λ	末尾に Load	直後に PreLoaded

Table 2: Interface Register への Promotion 条件と処理内容

この操作を単純化するために、Caller → Callee 間での引数・返値の binding を表す仮想命令 arg-Φ, ret-Λ を用いる (Table1)。Interface Register Promotion に先だって、最適化対象となる関数の引数・返値は arg-Φ, ret-Λ の形式に変換されているものとする。

3.2 Register Promotion for each Procedure

関数毎の Register Promotion は基本的に Lazy Code Motion であるが、Safe, Avail の計算の際、Table1 に示した命令に対応している点が異なる。以下にその手順を示す。

1. メモリアクセス命令の選択
2. 命令間の依存調査
3. Load, Store に関する Safe, Avail の計算
4. アクセス命令の再配置
5. 1 〜 4 を繰り返す

1 メモリアクセス命令の選択　関数内に存在する、最適化されていないメモリアクセス命令を 1 つ選択する。効果の高い命令を優先して Promotion するために、プロファイルデータなどを利用して実行回数の多い命令から順に処理する。

2 命令間の依存調査　ポインタ解析データを元に、手順 1 で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 (Same-Load, Same-Store) 以外に、Load するかもしれない (Potential-Load) と Store するかもしれない (Potential-Store) を定義する。また、union 等で生じる型が異なるアクセスに関しては、Potential-Load, Potential-Store と同等に扱い、Register Promotion の対象から外す。

		Gen	Kill
Load	Safe	Load	Store P-Store
	Avail	Load Pre-Loaded Store	P-Store
Store	Safe	Store	P-Load P-Store
	Avail	Store Post-Stored	P-Load P-Store

Table 3: 提案手法で Gen, Kill に相当する命令
(P-Load は Potential Load の略)
(P-Store は Potential Store の略)

3 Load, Store に関する Safe, Avail の計算
本手法における、Safe, Avail の算出法を Table3 に示す。Load, Store の Safe, Avail はそれぞれ Partial Redundancy Elimination における Gen と Kill を適宜差し替えることで求める。尚、Store に関しては Load と上下逆の解析を行う。

4 アクセス命令の再配置　通常の Register Promotion と同様に、引数・返値レジスタへの Promotion も、Load, Store の Safe, Avail 属性に応じて 4 種類の Promotion を適用する。Table2 に、4 種類の Promotion に関して、Promotion

するための条件、その際追加する Interface 命令、関数内に配置するアクセス命令、CallSite に配置するアクセス命令を示す。尚、4 つの条件のいずれにも該当しない場合は、通常の Register Promotion を行う。

本手法では CallSite に、関数内での挙動を示す命令を適切に配置することにより、関数間 Register Promotion は関数内の Register Promotion に帰着させる。次節で、その様子を例を用いて説明する。

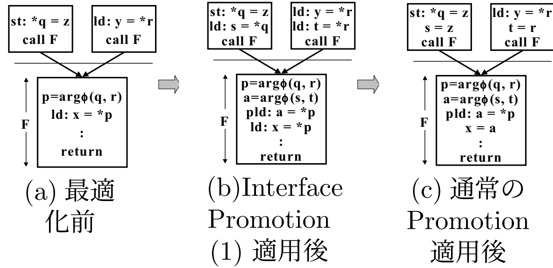

Figure 4: Interface Register Promotion Type(1)

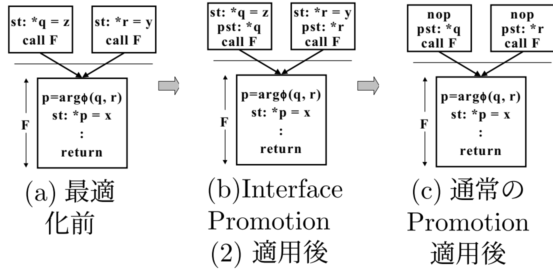

Figure 5: Interface Register Promotion Type(2)

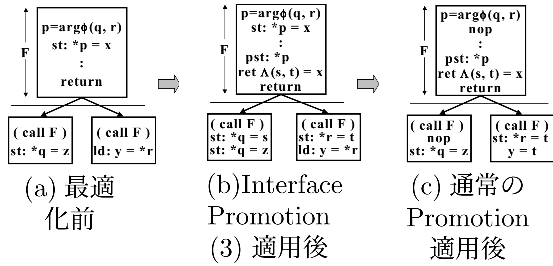

Figure 6: Interface Register Promotion Type(3)

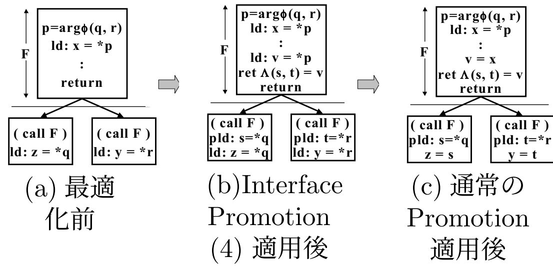

Figure 7: Interface Register Promotion Type(4)

名前	module	proc	line
compress	2	24	1934
li	25	357	7597
ijpeg	88	473	29117
go	36	372	29629
m88ksim	119	252	19092
bzip2	2	74	4649
gzip	14	112	8605
mcf	11	26	2412
twolf	75	191	20459
vpr	20	272	17729

Table 4: 評価に用いたベンチマークプログラムの特徴
(モジュール数・関数の数・行数)

3.3 Example

Figure4(a) は複数の CallSite から関数 F が呼び出されていて、ポインタ引数 p を介して冗長なメモリアクセスが行われているプログラムを示している。このプログラムに Table2 の (1) の Promotion を適用すると、関数の先頭に arg-Φ と Pre-Loaded が、CallSite の直前に Load がそれぞれ追加され Figure4(b) のように変形される。この後各関数に通常の Register Promotion を適用することで、一時的に増加したメモリアクセス命令が除去され、Figure4(c) の形に変化する。Pre-Loaded は実際にはメモリにアクセスしないので、この操作によりメモリアクセス回数が削減されている。

他の (2)〜(4) の Promotion に関しても同様の処理となる。それぞれの具体例は Figure5, Figure6, Figure7 に示す。

4 Evaluation

4.1 Environment

評価にあたって、最適化 C コンパイラ newcc[9] の alpha 21264 backend に対して簡単な Pointer Analysis, 簡単な Interprocedural Register Allocation と提案手法 Interface Register Promotion を実装した。その特徴を Figure8 に示す。

評価には SPECint95 と SPECint2000 の中から Table4 に示すプログラムを用いた。これらに対し、関数内の Register Promotion と Interface Register Promotion を行い、コード中の Load, Store の実行回数を測定した。尚、メモリアクセスの領域毎に Load, Store の削減量を議論するため、Table5 に示す 4 種の 領域を区別している。

Visible Scalar : Global Scalar 変数や Stack Scalar 変数へのアクセス
Arg + Const : ポインタ引数 + 定数 へのアクセス
Spill : Register Allocation によって生じる Spill
Others : 配列など、その他のアクセス

Table 5: グラフ中のメモリアクセス領域の区別

- Pointer Analysis
  - flow insensitive
  - context sensitive
  - global 変数と stack 変数を解析
- Interprocedural Register Allocation
  - GA 的なアルゴリズム
  - Profile を使用し、Spill 数見積もりが削減する方向に遷移
- Interface Register Promotion
  - Lazy Code Motion
  - Profile を使用し、投機的なコード移動も行う

Figure 8: 実装した最適化の特徴

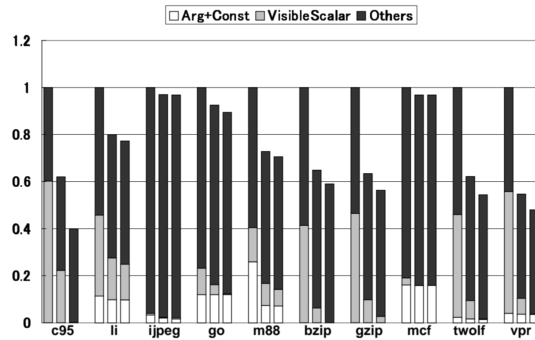

Figure 9: Register Promotion の Load に対する単体性能
(左から 最適化無し、関数内 RP、提案手法)
(縦軸は未最適化時を 1 とする Load 数を示す)

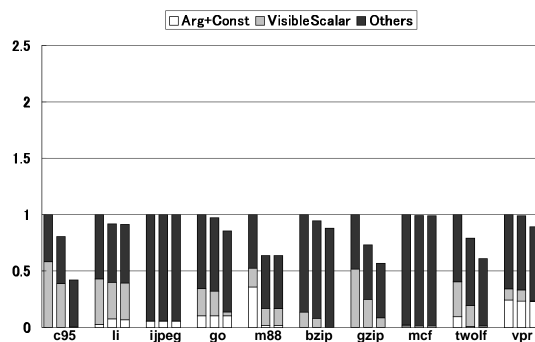

Figure 10: Register Promotion の Store に対する単体性能
(左から 最適化無し、関数内 RP、提案手法)
(縦軸は未最適化時を 1 とする Store 数を示す)

4.2 Access Elimination via Register Promotion

Register Promotion の単体性能を Figure9 と Figure10 に示す。図中の棒グラフは 3 本で 1 つのアプリケーションに対応し、それぞれ左から、最適化前、関数内の Register Promotion、Interface Register Promotion を意味する。また、縦軸は最適化前を 1 とする相対メモリアクセス数である。

今回の実装では Visible Scalar と Arg+Const のみが最適化対象となっているため、その他の領域へのアクセス割合が大きい ijpeg, go, mcf に対しては、Register Promotion によるアクセス数削減の度合いが小さい。

しかし、Visible Scalar の割合が多いアプリケーションでは、提案手法によるメモリアクセス数削減効果が大きく現れ、compress の Load で 60%、Store で 58% 削減できている他、vpr でも 50% の Load が削減されている。li, m88ksim に関しては Visible Scalar が少なからず残っているが、これは Library 呼び出しの影響である。特に li に於いては、setjmp, longjmp といった特殊な標準ライブラリが多用されており、最適化の機会が大きく損なわれてしまった。

また Arg+Const に対しては目立ったアクセス数の減少が見られない。この点に関しては、ポインタ解析の精度不足のためではないかと考えている。

4.3 Access Elimination via Register Promotion / Allocation

次に、実際のメモリアクセス削減量を評価するため、関数間 Register Allocation と合わせた場合のメモリアクセス数を測定した。この際、不用意に Register Promotion を行うと Register 負荷が高まり、Register Allocation 時に Spill が多発してしまうため、Register Promotion に Table6 に示す制約条件を設けた。

この制約は Table7 に示した alpha 21264 における整数レジスタの使用ルールに基づいており、各位置で生きている (レジスタに保持する必要のある) 変数の数を “Available Register 数” に制限し、Library や Dynamic Call を越えて生きている変数を “Callee-Save Register 数” に制限した。また、それ以外の関数呼び出しを越えて生きる変数や引数・返値の数は “Available Register

各位置で生きている変数の数	≤ 28
引数の数	≤ 14
返値の数	≤ 14
Library, Dynamic Call を跨ぐ変数の数	≤ 7
その他の CallSite を跨ぐ変数の数	≤ 14

Table 6: Interprocedural Register Allocation のために加えた heuristic

Available Register	28
Callee-Save Register	7
Caller-Save Register	21

Table 7: alpha 21264 の Conventional Rule
(各用途に使用可能な Integer Register 数)

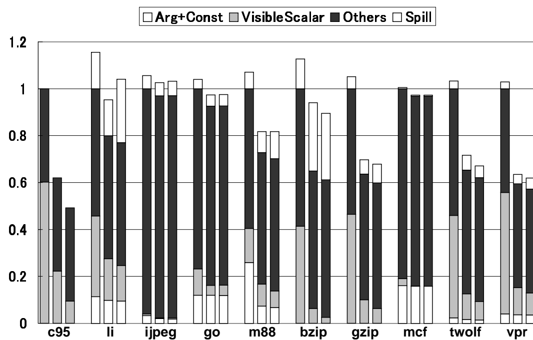

Figure 11: Register Promotion の Load に対する全体性能
(左から 最適化無し、関数内 RP、提案手法)
(縦軸は未最適化時を 1 とする Load 数を示す)

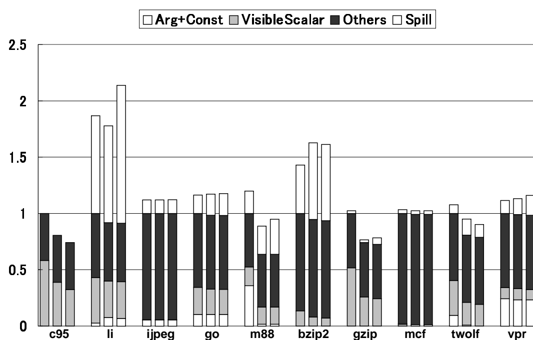

Figure 12: Register Promotion の Store に対する全体性能
(左から 最適化無し、関数内 RP、提案手法)
(縦軸は未最適化時を 1 とする Store 数を示す)

数の半分" に制限した。

関数内 Register Promotion と提案手法を比較すると、compress95 に関しては提案手法の効果が大きく現れており、Visible Scalar への 13% の Load , 6% の Store 削減に成功している。また、m88ksim, bzip2, gzip, twolf, vpr の Load で 3〜5%、twolf の Store で 5% のメモリアクセスを削減できている。しかし li の Load と、多くのアプリケーションの Store で、提案手法は 関数内 Register Promotion よりも多くの Spill を発生させてしまい、結果的に総メモリアクセス数が増加してしまっている。これは Spill を抑えるための heuristic が適切でなかったためと考えられる。今後は、より優れた Spill 抑制手法を検討する必要がある。

5 Conclusion

本稿では、関数間で Register Promotion を行う手法を再提案した。また、SPECint ベンチマークのいくつかのアプリケーションに対して提案手法を評価し、単体性能で最大 60% の Load と 58% の Store を削減できることを示した。そして Interprocedural Register Allocation と合わせた総合性能で、最大 50% の Load と 26% の Store を削減できることを示した。

今後の課題としては、ポインタ解析の精度を向上させる必要がある。また、Spill を避けるための制約に関しても再検討が必要だと考えている。

References

[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] 服部直也, 飯塚大介, 坂井修一, 田中英彦. コンパイラによるロード・ストアユニット負荷の軽減. *HPC-82*, pp. 107–112, 2000.

[9] 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.