自動的な命令合併を行う覗き穴最適化器の設計とプロトタイプ実装

Design and Prototype Implementation of Peephole Optimizer for Automatic Instruction Coalescing

佐原 聡一郎 † 佐々 政孝 †
Soichiro SAHARA Masataka SASSA

† 東京工業大学 大学院情報理工学研究科 数理・計算科学専攻
Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology
{sahara2, sassa}@is.titech.ac.jp

コンパイラのコード生成器が中間コードを目的コードに変換する際，冗長な命令や最適でないコードが出やすい．これらを最適化するためにしばしば行われるのが覗き穴最適化である．本研究の覗き穴最適化の特徴は，対象機械の命令の意味記述から，目的コードの命令列を，命令の意味を陽に表す中間コードに変換し，その上で隣接する 2 命令の合併を行う覗き穴最適化をかけ，その後再び意味記述を用いて目的コードに逆変換するという点にある．中間コードに変換することにより，命令合併は機械に依存することなく，自動的に行うことが可能となった．また，最適化の効果をあげるためにいくつかのデータフロー解析も行った．この最適化により，冗長な命令列の除去や無用命令の除去などが行えた．

1 はじめに

中間コードから目的コードに変換する際，中間コードの各命令ごとにコード生成を行うと，冗長な命令や最適でないコードが出ることが多い．これらを最適化するためにしばしば行われるのが，覗き穴最適化である．覗き穴最適化は，対象機械に依存した最適化も行いやすいという利点があるが，一方で，機械ごとに命令セットが異なるため処理が複雑になりやすいという問題もある．

本研究では，これらの問題を解決するために，自動的な命令合併を行う覗き穴最適化器を設計し，プロトタイプを実装した．自動的な命令合併の基本的な着想は，Davidson らによって提案された覗き穴最適化器 [1, 2, 3] をもとにしている．最適化の手法は，対象機械の命令の意味記述から，目的コードの命令列をセマンティクスを陽に表す中間コードに変換し，その上で隣接する 2 命令の合併を行う覗き穴最適化をかけ，その後再び意味記述を用いて目的コードに逆変換するというものである．中間コードに変換することにより，命令合併は機械に依存することなく，自動的に行うことが可能となった．命令の合併についての詳細は 2 節で述べる．

本研究では，最適化の効果をあげるためにさらにふたつのデータフロー解析を取り入れ，その結果の情報を用いて最適化を行った．解析の詳細は 3 節で述べる．

また，最適化の効果をベンチマークにより調べるため，COINS[4] コンパイラに組み込んで実験を行った．ベンチマークは SPEC CINT2000[5] を用いた．結果と考察は 5 節で述べる．

2 命令合併による最適化

本節では，隣接する 2 命令の自動的な合併による最適化について述べる．ここでいう命令合併とは，もとの 2 命令と等しい効果をもつ 1 命令を作ることをいう．合併された新命令の実行コストが，もとの 2 命令の実行コストより低ければ，もとの 2 命令を新命令で置換することで，実行速度の改善が期待できる．また，命令の絶対数が減るので，コードスペースの節約の効果も期待できる．

本研究では，最適化の処理の簡潔化および他の機械への移植性を考えて，目的コードの命令列を対象機械の命令のセマンティクスを陽に表す中間コードに変換し，その上で最適化を行った．以降では，説明のための例は SPARC を対象機械としている．

2.1 前提条件

隣接する 2 命令を合併するためには，ひとつ目の命令の後にふたつ目の命令が間をはさまずに実行されることが確実に保証されなくてはならない．そうでなければ，命令の合併によって，元の意味が変わってしまう可能性が生じてしまうからである．よって，合併は基本ブロック内で行う必要がある．本研究ではこの保証のために，命令列を基本ブロックに分割し，制御フローグラフを構築した[1]．

2.2 最適化のアルゴリズム

前提条件で述べたとおり，命令合併の最適化は基本ブロック内で行う．基本ブロック内の命令列を下から上へ逆順に見て，以下の処理を繰り返す．合併後の命令も新たな合併の対象とすることで，もれなく合併を試みている．なお，以下での命令は，中間コードの命令を表す．

1. 注目している命令を i_1，i_1 の下の命令を i_2 とする．
2. i_1，i_2 を合併した新しい中間コード i_{new} を仮につくる．
3. i_{new} が対象機械上で実現可能な命令で，かつ，もとの 2 命令より実行コストが低ければ i_1，i_2 を i_{new} で置換する．
4. 置換が起きたならば注目する命令を i_{new} にする．そうでないならば注目する命令を i_1 のひとつ上の命令とする．

合併後の中間コード i_{new} が対象機械上で実現可能かどうかを調べるのは，中間コードを目的コードに戻す逆変換の機能を使えば可能である．すなわち，逆変換に成功すれば実現可能ということであり，失敗すれば実現不可能ということになる．逆変換は，対象機械の命令の意味記述を用いて統一的な処理として行うことができるので，命令合併の最適化全体は機械非依存な処理として行うことが可能となる．

命令合併 i_1，i_2 がこの順で実行されるとき，命令合併は次の 3 ステップで行われる．

1. i_2 で使用される変数のうち，i_1 で定義されている変数があれば，その定義の右辺で置き換える．
2. ふたつの命令を連結する．連結された命令は並列に評価されるものとする．連結の結果，同じ変数に 2 回代入が起こる場合は，命令 i_2 にあった代入文以外を除去する[2]．
3. 合併した命令に，代数的最適化や無用文除去などの最適化を施し，簡潔な形にする．

例 以下に実際の命令合併の例を示す．

$$i_1\colon\ R[2] = R[1] - 2$$
$$i_2\colon\ icc = R[2] - 0$$

$R[n]$ は整数レジスタ，icc は整数型コンディションコードを表す．この 2 命令は，SPARC では SUB 命令，CMP 命令である．ステップ 1 の後，以下のようになる．

$$i_1\colon\ R[2] = R[1] - 2$$
$$i_2\colon\ icc = (R[1] - 2) - 0$$

ステップ 2 の後，以下のようになる．カンマは，文が並列に評価されることを表している．

$$i_{new}\colon\ R[2] = R[1] - 2,\ icc = (R[1] - 2) - 0$$

ステップ 3 で式の最適化を行った後，以下のようになる．

$$i_{new}\colon\ R[2] = R[1] - 2,\ icc = R[1] - 2$$

これは，SPARC では SUBCC 命令で実現可能である．もとは SUB，CMP の 2 命令だったのが，合併により同等の効果をもつ，より実行コストの低い SUBCC 1 命令になった．

3 データフロー解析

2 節で示した，命令合併による最適化の効果をあげるために，本研究では 2 種類のデータフロー解析を行い，その結果を用いて最適化を行った．2.1 節で述べたように，最適化のために制御フローグラフを構築しているので，大域的なデータフロー解析も比較的簡単に実現することができた．解析の方程式やアルゴリズムは文献 [6] を見られたい．

3.1 生存変数解析

生存変数解析の情報があれば，命令合併のステップ 3 においてより多くの無用文除去を行うことがで

[1] 基本ブロック分割では，SPARC の delay slot や annul bit を正しく扱うよう工夫した．

[2] 命令 i_1 の代入文は無用コードである．

き，合併後の中間コードが対象機械上で実現可能な形となる可能性が高くなる．例えば，以下の 2 命令の合併を考える．なお，$M[addr]$ は $addr$ 番地のメモリを表す．

i_1: $R[1] = 0$
i_2: $M[R[2]] = R[1]$

合併のステップ 1，2 により以下の中間コードができる．

i_{new}: $R[1] = 0, M[R[2]] = 0$

ここで，$R[1]$ が生存変数ではないということが解析によってわかっていれば，ステップ 3 で i_{new} は次のように変形される．

i_{new}: $M[R[2]] = 0$

代入文 $R[1] = 0$ が取り除かれる前の i_{new} は SPARC では実現できない命令であるが，取り除いた後の i_{new} ならば %g0 レジスタを用いた ST 命令として実現可能である．

本研究では，生存変数解析の対象を，中間コードにおけるレジスタ変数とコンディションコードを表す変数 (icc) とした．

3.2 ビット情報伝播

命令合併をさらに促進するために，本研究では生存変数解析の情報の他に，変数ごとのビット情報を計算した．これは簡単にいうと，定数伝播のビット版のようなものである．なお，ここでいう変数とは，生存変数解析のときと同様，中間コード上でのレジスタ変数とコンディションコードを表す変数のことである．

例えば，次のような命令があったとする．

i_1: $R[1] = R[2] << 10$

$R[2]$ はレジスタ変数なので，普通コンパイル時には値がわからないが，左に 10 bit シフトする演算により，$R[1]$ の下 10 bit は 0 であることが保証される．この後も，$R[1]$ が再定義されるまではこの特徴は変わらない．このようなビットの特徴を伝播させるのが，計算の目的となる．いうなればビット伝播である．これは，定数伝播をより広範囲に一般化したものとみなすこともできる．

本来ならば制御フローグラフを生かして，大域的なビット伝播の計算を行いたかったが，正確に行うのは困難だったため，本研究では基本ブロック内ごとに局所的に計算した．すなわち，ブロックの入口での全ての変数の全てのビットは不明な値として計算を行った．

計算の方法は，基本ブロック内の中間コードの命令列を上から順に見て，以下の処理を繰り返すことで実現できる．なお，基本ブロックの入口では全ての変数のビット情報は不明として表を初期化しておく．

1. 命令中のすべての代入文について，現在のビット情報の表を使って右辺の式のビット情報を計算する．
2. ビット情報の表の，代入文の左辺の変数の項目を，計算したビット情報で置き換える．

この解析によって，例えば SPARC では次のような 3 命令の並びの i_1，i_2 が合併可能となった．

i_0: $R[1] = (const >>> 10) << 10$
i_1: $R[1] = R[1] \mid (const \ \& \ 1023)$
i_2: $R[2] = M[R[1]]$

i_1，i_2 を合併すると以下の中間コードができる．

i_{new}: $R[2] = M[R[1] \mid (const \ \& \ 1023)]$

ここで，ビット情報を解析すると，i_0 で定義された $R[1]$ の下 10 bit は 0 であることがわかる．よって，下 10 bit が 0 である $R[1]$ と，下 10 bit 以外が 0 である $const$ & 1023 とのビット毎の論理和は，加算に置き換えられるので，i_{new} は次のように書き換えられる．

i_{new}: $R[2] = M[R[1] + (const \ \& \ 1023)]$

論理和演算のままでは SPARC では実現できない命令であったが，加算に置き換えたことで SPARC の $[reg + const13]$ というアドレッシングモードを利用した LD 命令として実現できる．

4 実装

本節では，設計した覗き穴最適化器の実装について述べる．本研究では，最適化器は Java のプログラムとして実装した．

まず命令合併の対象についてだが，整数型演算命令同士のみに限定して実装した．原理的には浮動小数点数型演算命令も合併可能であると考えられるが，命令合併はその性質上定数の畳み込みが起こること

があるので，浮動小数点数型命令を合併の対象としてしまうと，対象機械上での浮動小数点数の計算を正確にシミュレートする必要がある．これはかなりの困難が予想されたので，今回は実装を見送った．

次に対象機械についてだが，他の機械への移植性を考え，最適化を中間コード上で行うよう設計したことは 2 節でも触れたが，今回，実際に実装を行ったのは SPARC のみである．

5 実験と考察

本節では，本研究の最適化の効果を実験により検証，評価する．本研究で実装した最適化器は，アセンブリ言語を入力とするが，アセンブリ言語のプログラムの出力のために COINS コンパイラを用いた．実験は，表 1 に示す実行環境で行った．

Architecture	Superscalar SPARC Version 9
CPU	750MHz UltraSPARCIII × 2
Cache	1 次データキャッシュ 64KB 1 次命令キャッシュ 32KB 外部 2 次キャッシュ 8MB
Memory	1GByte
OS	SunOS 5.8
COINS	1.2.1 リリース版

表 1: 実験環境

比較の対象として，プログラムに対して次のような 4 種類のコンパイルを行い，A と B，C と D を主な比較対象とした．

A: COINS 最適化なし

B: COINS 最適化なし → 本研究の覗き穴最適化

C: COINS 各種最適化[3]

D: COINS 各種最適化 → 本研究の覗き穴最適化

実験に用いたプログラムは，SPEC CINT2000 に収録されているベンチマークのうちの，164.gzip, 175.vpr，181.mcf，197.parser，256.bzip2，300.twolf の 6 つである．

静的命令数の比較を表 2 に，実行時間の比較を表 3 に示す．なお，表 3 での SPEC ベンチマークの入力は ref で，実行時間は 3 回の測定の中央値を示している．

	A	B	C	D
164.gzip	16114	14446	15587	14134
175.vpr	47637	44895	44219	41562
181.mcf	2721	2605	2698	2619
197.parser	27712	26710	27737	26747
256.bzip2	8865	8181	8227	7656
300.twolf	80499	70830	71358	63743

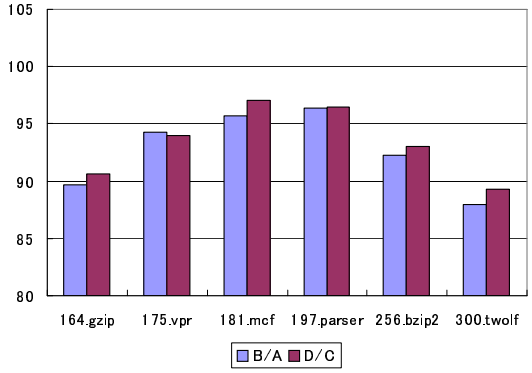

表 2: 静的命令数の比較

表 2 を見てみると，平均して 7%程度の命令数が削減できている．命令の削減に関して言えば，十分な効果といえるだろう．

表 2 と表 3 を見比べてみると，命令数の減少と実行時間の減少の割合にはおおよその相関関係が見られ，平均しても 5%前後の実行時間の改善が見られた．

しかし，不可解な結果もでている．2 節で述べたように，命令の合併は実行コストが局所的には増えない場合にのみ行うこととしたので，全体の実行コストも増えないと思われていたが，表 3 を見てみると，181.mcf のコンパイル B における実行時間が本研究の最適化により有意に遅くなっているのがわかる．

原因について確かなことは示せないが，スーパースカラーマシンにおける，命令キャッシュラインとループ先頭番地との関係が主に影響していると考えられる [7]．命令フェッチの都合上，ループの先頭番地のアドレスが 32 byte 境界などきりのよいアドレスになっている方が，実行の効率がよくなったりする．おそらく，本研究の最適化により命令数が変化

[3]最適化は全て LIR 上でのものである．その内容は，SSA 変換 → 3 番地コードへ変換 → 共通部分式除去 → 定数伝播と畳み込み → ループ不変式移動 → 帰納変数の演算の強さの軽減と判定の置き換え → ループ不変式移動 → 定数伝播と畳み込み → 共通部分式除去 → 無用コード除去 → SSA 逆変換 (Sreedhar 法) である．

	A	B	C	D
164.gzip	872	801	835	772
175.vpr	1101	1044	805	771
181.mcf	471	491	473	464
197.parser	915	873	892	866
256.bzip2	1069	1018	891	853
300.twolf	1218	1171	1123	1065

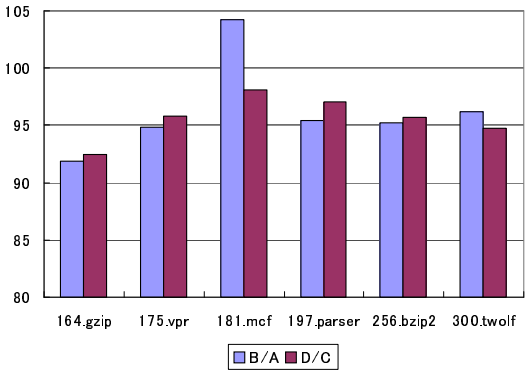

表 3: 実行時間の比較 (単位:秒)

し，命令のメモリ上の配置が変わったことで，このような現象の影響が出てきたものと思われる．

ただ，実行時間が増加してしまったのは，6 つのなかで一番規模の小さいものであったことから，他の要因による影響が強く出た可能性もある．今回の実験だけではデータの量が不足しているため断定はできないが，大きなプログラムほど他の要因の影響が相対的に弱まっていると仮定すれば，本研究による最適化の効果はおおよそ 5%前後の改善があったと言えるかもしれない．

6 おわりに

本研究では，対象機械の命令の意味記述を利用して，自動的な命令合併を行う覗き穴最適化器を設計した．より多くの命令を合併するために 2 種類のデータフロー解析を行い，その情報を利用した．また，SPARC の目的コードを扱えるよう，プロトタイプ実装し，COINS コンパイラと組み合わせて最適化を行った．

SPEC CINT2000 を用いた評価実験では，命令数の削減という観点からみると，本研究の最適化は確かな効果をあげていた．実行時間の改善に関しては，おそらく他の要因の影響が強く作用し，一部改善の見られないものもあったが，全体的に改善されたと考えるのが妥当である．いずれにせよ，さらなる実験でデータを集めたい．

謝辞

本研究の一部は，文部科学省科学技術振興調整費，科学研究費，および財団法人栢森情報科学振興財団の補助を受けた．

参考文献

[1] Jack W. Davidson and Christopher W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Trans. Program. Lang. Syst.*, Vol. 2, No. 2, pp. 191–202, 1980.

[2] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst.*, Vol. 6, No. 4, pp. 505–526, 1984.

[3] Jack W. Davidson and Christopher W. Fraser. Automatic generation of peephole optimizations. *SIGPLAN Not.*, Vol. 39, No. 4, pp. 104–111, 2004.

[4] COINS Project. Coins project home page. http://www.coins-project.org/.

[5] SPEC. Standard performance evaluation corporation home page. http://www.spec.org/.

[6] 佐原聡一郎. 移植可能な覗き穴最適化器の設計と実装. 東京工業大学 情報科学科 卒業論文, 2005.

[7] Sun Microsystems. Inc. *Ultra SPARC III Cu User's Manual*, 2004. http://www.sun.com/processors/manuals/USIIIv2.pdf.