実行サイクル数予測に基づく 大域的命令スケジューリングの実装と評価

服部直也　荒木拓也[†]　坂井修一　田中英彦
東京大学工学系研究科

概要

大域的スケジューリングはベーシックブロックを超えた命令移動を行い, プロセッサの並列実行能力を支援する技術である. 命令移動にはコストが伴うため, 全ての移動を行うことはできず, 有効な移動を選択する必要がある. これまでの手法では, 命令移動の有効性判断・優先順位付けの際に, (1) ベーシックブロックを超えたデータ依存あるいは, (2)ALU, FPU といったプロセッサの持つ機能ユニットの種類と数に対する配慮, が不足していた. 本稿ではこれらの問題点を解決する手法を提案し, 本手法はスケジューリングを行わない場合と比べ 10% 程度の速度向上が得られることを確認した.

Implementation and Evaluation of Global Instruction Scheduling Based on Execution Cycle Estimation

Naoya Hattori　Takuya Araki　Shuichi Sakai　Hidehiko Tanaka
Graduate School of Engineering, University of Tokyo

Abstract

Global instruction scheduling is a technique to improve parallel execution in processors by moving instructions between basic blocks. As processor resource is limited, we need an efficient selection algorithm to choose which instructions to move. Previously proposed selection algorithms have limitations in (1) Not considering data dependency between instructions in different basic blocks, or (2) Not considering either type or number of execution units available in processors. In this paper, we propose a new scheduling technique which overcomes the limitations described above. Compared with non-scheduled code performance, we get 10 % improvement.

[†]現 NEC

1 はじめに

近年のプロセッサはクロック周波数の向上だけではなく，複数命令を並列実行することで性能向上をはかってきた．コンパイラはそのサポートとして，プログラムに内在する並列性を抽出するために命令スケジューリングと呼ばれる命令並び替え操作を行っている．命令スケジューリングは大きく2種類に分けられ，ベーシックブロック内で命令再配置を行う局所スケジューリングと，ベーシックブロックを超えた命令移動を行う大域的スケジューリングがある．

大域的スケジューリングでは，ベーシックブロックを跨いで命令を移動し，機能ユニットが使われていないサイクル (空きスロット) を減らすことで高速化をはかる．しかし大域的な命令移動には

1. プログラムの意味を保持するためにレジスタを消費する (レジスタリネーミング)．
2. 移動先ブロックの空きスロットを消費する．

というコストが伴うため，移動できる命令には限りがある．そのため，有効な命令移動を選択しなければならない．また，時間的制約から一度行った命令移動を取り消すことはできず，有効な命令移動を優先する順序付けが必要になる．これらの処理には分岐履歴情報を用いる．情報としてはプロファイルデータと呼ばれる，実際にプログラムを動作させて採取したデータか，プログラム構造から予測したデータを用いることになる．

これまでのスケジューリングの有効性判定・優先順位付けには，

- 命令の実行頻度のみを重視し，ベーシックブロックを超えたデータ依存を考慮していない．

あるいは

- データフローグラフを用いているため，プロセッサ内に有する機能ユニットの種類や数を考慮していない．

という問題点があった．本稿ではこれらの問題を解決する大域的スケジューリング手法について述べる．

2 関連研究

分岐の両方のパスから命令移動を行う手法として，まずパーコレーションスケジューリングが挙げられる．この手法は命令移動を上方のみに限定し，対象命令を上げられるだけ上に無条件で移動する．この手法は，移動の優先順位付けを行わないため無駄な移動で資源を浪費する可能性があり，また移動の有効性判定を行わないため，並列実行できない命令を移動して速度低下を招く恐れがある．そこでこの欠点を改善する手法が提案されている．

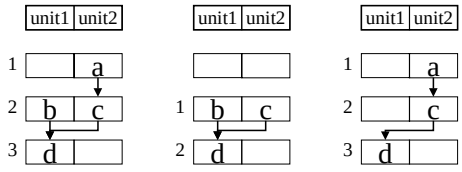
セレクティブスケジューリング [3] は資源予約表を用いる．資源予約表とは，各命令がどのユニットへ何サイクル目に発行されるかを示す表である．図1などの資源予約表は横方向が各機能ユニット，縦方向がサイクルを表している．この手法では上流の空きスロットに対して，そこへ移動可能な全ての命令を下流ブロック全てから探し，その中で最も実行確率の高い命令を移動する．しかし実行確率のみで判断しているため，上流の命令から順に移動すれば，下流の命令を移動しやすくなるという性質を考慮していない (詳細は 3.1で述べる)．

小松らのスケジューリング [6] はデータ依存と制御依存を対等に扱うことで，プログラム中の依存関係を 1 つのグラフに表す．このグラフを基に，下流ブロックのクリティカルパストップの命令を，上流ブロックのクリティカルパスを伸ばさないように移動する．この手法では実行確率の高いトレース順にスケジューリングを進めることで，ベーシックブロックを越えた依存を考慮している．しかし，クリティカルパスを見ているだけなので，ユニットの種類や数に関しては考慮されておらず，これでは実行時間のよい見積もりを得ることはできない (詳細は 3.3で述べる)．

3 新しいスケジューリング手法の提案

大域的命令移動の際にサイクル数減少の鍵となるのは，移動元ブロックのサイクル数が減少するか否かである．命令移動によって並列実行命令数を増やしたとしても，移動元の命令がそのベーシックブロックのボトルネックになっている命令でなければ，サイクル数の減少は望めない．例えば図 1(a) の場合，命令 a が無くなればサイクル数が減少するが (図 1(b))，命令 b が無くなってもサイクル数は変わらない (図 1(c))．この例の a のような命令をここではボトルネックと呼ぶ．本手法ではまず移動元のブロックのボトルネックを探し，次にその命令の受け入れ先を探すというアプローチを採る．

また，少しずつ移動を繰り返していくと分岐を越えた数だけレジスタを消費する可能性があるた



(a) 移動前　(b) 命令 a を移動　(c) 命令 b を移動

図 1: 移動元ブロックの高速化

各図は資源予約表, a, b, c, d は命令, 矢印は依存関係を示す
横方向が各ユニット, 縦方向はサイクルを表す

め, 命令移動の際は移動可能な最上流へ移動する.

我々のスケジューリングでは優先順位付けとして,

phase 1 ベーシックブロックの優先順位付け

phase 2 ブロック内命令の優先順位付け

の 2 段階の処理を行う. また, 各移動の有効性を判断するために

phase 3 実行サイクル数の見積り

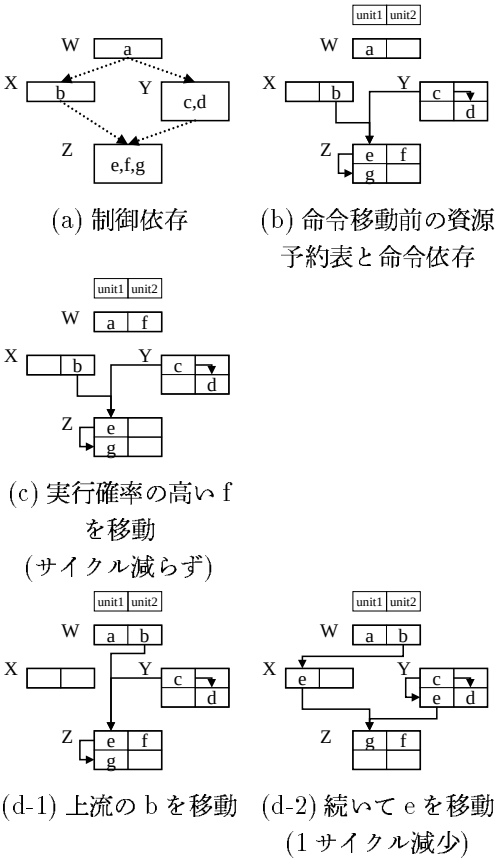
を行う.

1 においては, ブロックの実行頻度とブロックを越えた依存の双方を尊重し, 実行確率の高いトレースを優先する. 2, 3 においては, ユニットの種類と数を考慮し資源予約表を用いた解析を行う. 各処理の詳細は次節以降で述べる.

3.1 ベーシックブロックの優先順位付け (phase 1)

実行頻度の高いブロックは高速化の効果が大きく出るため, 優先して移動を行い高速化すべきである. しかし, 単純に実行頻度だけに着目する訳にはいかない.

例えば図 2(a) のような制御依存を持ち, 移動前の資源予約表が図 2(b) であるようなベーシックブロック群を考える. 実行頻度順に移動を行う場合はブロック W, Z が X, Y よりも優先され, この場合はブロック Z の命令 f をブロック W へ移動する (図 2(c)). これ以上移動できる命令はないのでスケジューリングは終了するが, 実行サイクル数は移動前と変わっていない. この場合は制御依存に従って上の命令から移動していくと, まずブロック Y の命令 b をブロック W へ移動し (図 2(d-1)), 次にブロック Z 命令 e をブロック Y とブロック X



(a) 制御依存　(b) 命令移動前の資源予約表と命令依存

(c) 実行確率の高い f を移動 (サイクル減らず)

(d-1) 上流の b を移動　(d-2) 続いて e を移動 (1 サイクル減少)

図 2: ブロックを越えた依存関係の例

へ移動する (図 2(d-2)). この操作により移動前と比べてブロック Z が 1 サイクル分の高速化された.

一般に命令にはベーシックブロックを越えたデータ依存があるため, 上流の命令を移動すると下流の命令が移動しやすくなるという利点がある.

そこで本手法では両者の間を取り, 実行確率の高いトレースから順に処理を進めていく. 分岐確率が図 3(a) であるブロック群の場合は, 図 3(b) のような順になる.

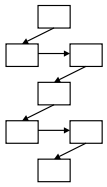
ただし多くの場合, 一般にループの内側から外側へは命令移動ができないため, ループを超えた依存は考えないことにする. このような場合は実行頻度の高いループを優先し, そのループ内のブロックに対して上記手法で順序付けする. 図 4(a) のような実行回数であるブロック群の場合は, 図 4(b) のような順で優先順位を付ける.

3.2 ブロック内での命令の優先順位付け (phase 2)

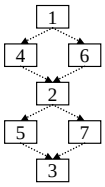
ベーシックブロック内には, その実行速度を低下させている命令がある. 例えば図 1では命令 a

(a) ブロックの実行回数

(b) **実行確率の高いトレース順 (1-trace)**

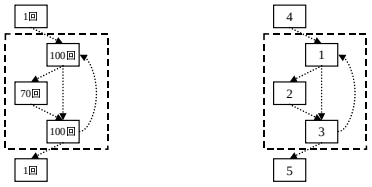


(c) 上から (1-topdown)



(d) 実行頻度順 (1-frequency)

図 3: ブロックの優先順位付け



(a) ブロックの実行回数　(b) ブロックの優先順位

図 4: ループを含む場合の優先順位付け

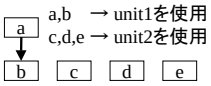
が実行サイクル数を増やしていると見ることができる．このような命令は，優先して移動する必要がある．

ボトルネックの判定にはプロセッサの機能ユニット種類と数が重要である．例えば，図 5(a) のような依存をもつ命令群を考える．データフローグラフを見ると，命令 a がサイクル数を増加させる原因であるように見える．しかし，unit2 を 1 つしかもたないプロセッサであれば，下詰め資源予約表 (図 5(c)) からわかるように，命令 c, d, e がサイクル数増加の原因となっている．そこで本手法では，「下詰め資源予約表の最上段に並ぶことのある命令」をボトルネックとして認識する．

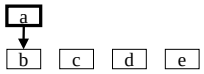
直接サイクル数が減少しない命令であっても，

1. 移動によってスロットが空く，
2. 早い時期に計算結果が得られるため，下流のブロックからの命令移動を可能にすることがある．

という効果があるため，直接サイクル数が減少し

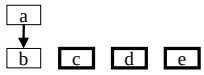


(a) 命令の依存関係



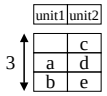
(b) データフローグラフからのボトルネック解析 (2-dataflow)

(c) 資源予約表



(d) **資源予約表からのボトルネック解析 (2-resource)**

(e) データフローグラフからのサイクル数見積もり (3-dataflow)



(f) **資源予約表からのサイクル数見積もり (3-resource)**

図 5: 移動すべき命令の検出とサイクル数の見積もり

ない命令も移動対象とする．

3.3 実行サイクル数の見積もり (phase 3)

大域的命令移動を行う上では，一部のブロックの実行時間の増減にとらわれず，全体の実行時間を減少に向かわせなければならない．この判断を行うには，実行サイクル数を見積もることが必要である．

我々の手法ではサイクル数の見積もりとして，

$$\sum(\text{各ブロックの実行に必要なサイクル数} \times \text{各ブロックの実行回数})$$

を用いる．図 6の場合は，

$$\text{見積もりサイクル数} = 1 \times 100 + 1 \times 70 + 2 \times 30 + 2 \times 100$$

となる．なお見積もりサイクル数は，分岐予測ミス，キャッシュミスはないものとして計算する．

ベーシックブロックのサイクル数見積もりに，データフローグラフを用いてしまうと，例えば図 5では 2 サイクルと見積もることになる (図 5(e))．

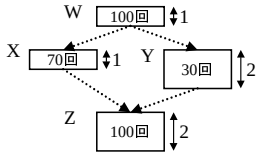


図 6: サイクル数見積もり

しかし, ユニットの種類と数を考慮すると 2 サイクルでの実行は不可能であることがわかる (図 5(f)). そこで, データフローグラフをのこのような問題点を解決するため, 本手法では資源予約表を用いた方法を採用する.

4 性能評価

4.1 評価条件

今回提案したスケジューリングの評価として, スケジューリングしたプログラムを UltraSPARC の実機上で動作させ, user time を測定した. UltraSPARC はスーパースカラプロセッサであるが, Out-Of-Order 発行をしないためスケジューリングの効果がそのまま現れる. スケジューリングは UltraSPARC 用に用意したポストスケジューラで行い, スケジューリングのベースとなるアセンブラソースは egcs ver 1.1 release に,

- スケジューリング,
- delayed-branch 最適化[1],
- peephole 最適化[2]

の 3 つ以外のオプションをつけて出力させた. 測定には Sun Ultra 1(UltraSPARC 167MHz, メモリ 96M) を用いた. なお今回の測定では, スケジューリングの最大性能を調べるために理想的なプロファイルデータを使用している.

測定項目として, 表 1に示したスケジューリングを用意した.

always とはポストスケジューラのオプションで, パーコレーションスケジューリングのような無条件な命令移動を行う base と local は局所, 大域スケジューリングの効果を見るために用意した. また, 1-topdown, 1-frequency, 2-dataflow, 3-dataflow により, phase1-3 で別のアルゴリズムを採用した場合の性能を調べる. なお, 比較対象以外の phase は proposed と同じものを用いている.

[1]局所スケジューリングの一種と判断した

[2]解析困難なコードを生成するため外した

• base	スケジューリングなし
• local	局所スケジューリング + delay slot 最適化
• always	無条件命令移動 (パーコレーション的)
• egcs	egcs のスケジューリング
• **proposed**	提案した手法
• 1-topdown	phase1 で上ブロックを優先 (図 3(c))
• 1-frequency	phase1 で頻度順に優先 (図 3(d))
• 2-dataflow	phase2 でデータフローグラフを使用 (図 5(b))
• 3-dataflow	phase3 でデータフローグラフによる見積もりを使用 (図 5(e))

表 1: 測定したスケジューリング

アプリケーション	入力
compress	bigtest.in
go	50 21 null.in
li	*.lsp
ijpeg	vigo.ppm
perl	scrabbl.pl scrabbl.in

表 2: 評価に用いるアプリケーション

4.2 結果

測定結果を表 4に, base からの速度向上率を表 5に示す. 表 4中の数値はアプリケーション終了までの user time (sec) である. スケジューリングはメモリアクセスの時間までは短縮できないため, 性能向上は単純な速度向上率では説明できないが, 目安として速度向上率を載せておく.

perl においてはどのスケジューリングを用いていも性能向上はほとんど見られなかった, これは perl のような並列性の低いプログラムに対しては, スケジューリングの効果が薄いことを示している. 以下では perl の項を除いて議論を進める. また, go の 1-frequency 他からの類推される値から大き

ユニット名	個数
ALU	2
FPU	2
Load/Store	1

表 3: UltraSPARC の備える機能ユニットの概要

condition	compress	go	ijpeg	li	perl
base	332	296	188	498	251
local	320	276	174	504	252
always	310	297	197	468	251
egcs	320	282	170	481	258
proposed	299	262	169	464	247
1-topdown	308	267	171	463	257
1-frequency	310	296	175	469	256
2-dataflow	301	261	170	464	244
3-dataflow	318	291	193	475	250

表 4: 測定結果

condition	compress	go	ijpeg	li	perl
base	1.00	1.00	1.00	1.00	1.00
local	1.04	1.07	1.08	0.99	1.00
always	1.07	1.00	0.95	1.06	1.00
egcs	1.04	1.05	1.11	1.04	0.97
proposed	1.11	1.13	1.11	1.07	1.02
1-topdown	1.08	1.11	1.10	1.08	0.98
1-frequency	1.07	1.00	1.07	1.06	0.98
2-dataflow	1.10	1.13	1.11	1.07	1.03
3-dataflow	1.04	1.02	0.97	1.05	1.00

表 5: 速度向上率

くずれているため議論から除外した．このずれの原因は調査中である．

egcs はプロファイルデータを用いていないため公正な比較ではないが，egcs の平均速度向上率が 1.06 なのに対し，本手法 (**proposed**) は 1.11 と優れた性能を出している．同様に，phase2 で 2-dataflow を用いる手法も，平均速度向上率が 1.10 であり，2-resource を用いた場合とほぼ変わらない．この理由としては，今回用いたアプリケーションは並列性に乏しく，図 5のような状況が稀だったのではないかと考えている．より並列性の高いアプリケーションでは phase2 によって差が現れると思われる．

phase1 に関しては，1-topdown の場合平均 1.09，1-frequency の場合平均 1.07 となり，確率だけで判断しては速度向上は望めないことおよび提案した手法で用いている，実行確率の高いトレース順の優先順位付けの有効性が示された．

phase3 に関しては，3-dataflow の場合は平均速度向上率が 1.02 と低く，無条件で移動を行った場合と同じような結果となっている．これより提案した手法で用いている，ユニットの種類と数を考慮する見積もりの有効性が示された．

5 まとめ

プロセッサの性能向上をはかるために，命令スケジューリング，特に大域的命令スケジューリングは重要な技術である．大域的スケジューリングでは，処理時間からの制約から一度行った移動を取り消すことはできない．そのため命令移動の取捨選択が必要になる．本稿では，理想的なプロファイルデータを用いて評価を行い，各種の優先順位付けとサイクル数の見積もりを用いた場合の最大性能を比較した．その結果提案した手法は，スケジューリングを行わない場合と比べて 10% 程度の速度向上が得られることを示した．

今後の課題としては以下の点が挙げられる．今回の評価では，ソフトウェアパイプライニング，テイルデュプリケーション，プレディケーション等の技術は用いていない．これらの技術を用いた場合の更なる高速化について研究する必要がある．また，今回の測定はプロファイルデータがあることを仮定しているので，プロファイルデータがない場合の性能評価が必要である．

参考文献

[1] *UltraSPARC^{TM} User's Manual.*

[2] Mike Johnson. *Superscalar Micropeocessor Design.* Prentice-Hall, 1991. 村上 和彰訳 スーパースカラ・プロセッサ, 1994.

[3] Soo-Mook Moon and Kemal Ebicioğlu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. *ACM Transaction on Programming Languages and Systems*, Vol. 19, No. 6, pp. 853–898, November 1997.

[4] David L. Weavar and Tom Germond. *The SPARC Architecture Manual Version 9.* SPARC international inc, 1994.

[5] 安藤秀樹. VLIW プロセサの性能を引き出すコンパイラ技術 (上). 日経エレクトロニクス, No. 663, pp. 163–180, 1996. 1996 年 6 月 3 日号.

[6] 小松秀昭, 古関聰, 深澤良彰. 命令レベルアーキテクチャのための大域的コードスケジューリング技法. 情報処理学会論文誌, Vol. 37, No. 6, pp. 1149–1161, 1996.

[7] 中谷登志男. VLIW 計算機のためのコンパイラ技術. 情報処理, Vol. 31, No. 6, pp. 763–772, 1990.

[8] 村上和彰. スーパスカラ・プロセサの性能を最大限に引き出すコンパイラ技術. 日経エレクトロニクス, No. 521, pp. 165–185, 1991. 1991 年 3 月 4 日号.