Modern Compiler Implementation in ML; 第20章

久野　靖*

1999.3.1

1　イントロ

□ 素朴な計算機は、一時に1つの命令しか処理しない。

- fetch → decode → read operand → ALU OP → write back

□ しかし現代のCPUは一時に多数の命令のさまざまな部分を実行している。

- 複数の命令について上記の各部を実行中
- 実行の開始待ち状態のものも

□ 通常、命令の流れは1つだけ。

- 複数のプログラムを同時に実行してるのではない。
- 命令列の中から隣接する命令群を並行して取り出し実行(命令レベル並列、ILP)。このおかげで近年のCPU性能は劇的に向上。

1.1　ILPのいろいろ

□ パイプラインマシン→連続する命令群を「流れ作業で」行う。前の命令のWBと次の命令のALUが同時、とか。

□ VLIWマシン→1つの命令語の中に複数の命令が詰め込まれていて並列に実行(依存関係がないことをコンパイラが保証しないといけない)。

□ スーパスカラマシン→「データ依存などがなければ」隣接する命令を並列実行(依存のチェックはデコードハードウェアで)。

- 依存関係が見つかったら順番に実行。
- ということは、依存関係があっても正しく実行されるが、コンパイラが依存関係を避けるように命令を並べると速く実行可能。

□ 動的スケジューリングマシン→命令を依存関係に応じて並べ替えながら実行する→コンパイラへの依存はより小さい。

*筑波大学大学院経営システム科学専攻

1.2　命令実行の制約

□ 命令実行の並列度が高いほど、プログラムの実行速度は速い。では「すべての命令を同時に(並列に)実行」できないのか?

□ 答えは、命令の実行に関する「制約」があり、これを守らなければならないから。

□ その制約を満たすような、最良の「スケジューリング」を探すことでプログラムを最適化できる。

1.3　制約の分類

□ データ依存:　命令Aが計算した結果を命令Bが利用→Aが終わらないとBが実行できない。

□ 機能ユニット:　乗算器がk_{fu}個しかなければ、その数以上の乗算命令を並列に実行できない。

□ 命令発行:　命令発行ユニットは最大k_{ii}命令しか並列に命令を発行できない。

□ レジスタ:　同時にはk_r個のレジスタしか使用できない。

□ 後半3つを合わせて「資源制約」「資源ハザード」と呼ぶ。

1.4　順序制約

□ パイプラインマシンでは「BはAより前に実行できない」にしても、「Bの一部分は実行できる」(例:フェッチ等)かも。

□ また、変数を名前替えすることで解消可能な「疑似制約」もある。

- Write-after-Write:　命令Aが場所Mに書き込み、Bも場所Mに書き込むなら、BをAより先にはできない。しかしAとBが別の場所に書き込むように変形すればOKになることも。
- Write-after-Read:　AがMから読みBがMに書き込むなら、BをAより先にはできない。しかし別の場所を使えばOK。

1.5 命令の資源使用

□ 各命令ごとに、そのサイクル数と各サイクルの使用資源を記述→ Fig20.1。

- 命令 A のサイクル i と B のサイクル j で同じ資源を共有→ A と B の $i-j$ サイクルずれた実行は不可→ Fig20.2。
- ただしマシンによっては複数の資源を持つ→その資源の範囲内であれば実行可能→すべての重なる命令を一緒に考慮する必要

1.6 命令のデータ依存

□ データ依存も同様に考える。

- 命令 A の結果は Write ステージでレジスタに書かれる→命令 B がそのレジスタを参照するなら、B の Read ステージは A の Write ステージより後である必要。
- マシンによってはバイパスを設けて A の演算サイクルの直後に B の演算が行えるようにしている→ Fig20.3。

2 資源制約なしのループスケジューリング

□ データ依存と資源制約双方を考慮した最適なスケジューリングは難しい (NP 完全)。

- NP 完全が駄目とは限らない (e.g. レジスタ採色) が…こちらは易しくない。

□ ます資源制約を考慮しないアルゴリズムを見てみる

- 実用にはならないが命令レベル並列性についてよく分かる

2.1 Aiken-Nicolau loop pipelining

□ 次の順番で処理

- 1. ループ展開
- 2. 各命令を可能な最も早い時点にスケジュール
- 3. 命令をループ数対時間でタブローにプロット
- 4. 命令グループとその傾きを見つける
- 5. 傾きを統一
- 6. 命令を畳み込む

□ 例としてプログラム 20.4a を使う。

- 各命令は 1 サイクル、並列発行数はいくつでも可能とする。

2.2 メモリ経由のデータ依存

□ store/fetch の最適スケジュールのため、メモリ経由のデータ依存を追跡する必要→とても難しい (see 19 章)。

- 20.4a の例を簡単に扱うため「スカラー置換」を使う。
- $V[i-1]$ をそれと同等である b に置き換える。
- その結果、各メモリ参照は (配列のオーバラップがないとして) 独立に。

□ 次にループ本体中の各変数参照について「この周回の値」か「前の周回の値」かを判別

- データ依存グラフ (20.4b) →点線は「ループに運ばれる依存」(20.5a)

□ 次にループを展開→依存グラフは DAG に (20.5b)。

- DAG は簡単にスケジューリング可能 (p.474)。
- これをタブローの形で表すと便利→ 20.6a。

□ 最初の数命令の後→タブローに決まったパターンが出現。

- 「cdgeh」→傾き 3、「abg」→傾き 2、「fj」→傾き 0。
- 周回数がある数以上ではまったく同じパターンの反復 (ここでは 4 以上)。

2.3 定理:

□ 定理

- ループに K 命令が含まれるなら、同一パターン反復の開始は K^2 周回以内。
- データ依存を保ったまま傾きの緩いものを急なものに並行に揃えることができる。
- 結果としてできるタブローは途中が全く同一の反復になる→パイプライン化されたループ本体。
- 以上の方法でスケジュールされたループは実行時間が最短。

□ 証明は参考文献に譲る

- 直観的には、展開後の DAG で最後の命令までのパス長が P なら、その命令はタブローでも P 番目のサイクルに来るから。

□ 最終的に 20.6b のタブロー→ 3 サイクルのパターン。

- このパターンが始まるのは 8 サイクル目→この前の部分はループのプロローグ、3 サイクルパターンが本体、最後の部分がエピローグ。
- 多命令発行のプログラム→ Fig20.7。
- j_i と j_{i+1} が同時に生きている→複数変数にして move を挿入→ Fig20.8。

□ これで 8 命令同時発行可能、4load/store 同時実行可能なマシンでの最適スケジューリングが完成。

□ ここでは各命令の所要時間 1 サイクルとしたが、複数サイクルに拡張するのは容易。

3 資源制約のあるループパイプライニング

□ 本物のマシンでは資源制約 (ロード/ストア機構、加算器、乗算器などの個数) があり、また同時発行できる命令数にも上限がある。

□ 実用的なスケジューリングアルゴリズムはこれらに対処する必要。

□ スケジューリングアルゴリズムに対する入力:

- スケジュールされるべきプログラム
- どの命令がどのパイプラインステージでどのリソースを使うかの記述 (図 20.1 のようなもの)
- マシンにおける利用可能な資源の記述 (上記の資源や同時発行可能な命令間の制約など)

□ 資源制約を持つスケジューリングは NP 完全問題→何とか「典型的」なケースでそれなりの結果をもたらす近似アルゴリズムを用いる

3.1 モジュロスケジューリング

□ 反復モジュロスケジューリング→最適ではないが実用的な資源制約を考慮したループスケジューリングアルゴリズム

□ 基本的にはバックトラックしながら制約を満たす (かつレジスタ割当て可能な) スケジューリングを探して行く。

□ すべての命令を△サイクルのループ本体に納める (プロローグ、エピローグは別途ある) → Aiken-Nicolau と同様。

□ 小さい△から始めて駄目なら 1 つずつ△を大きくしていく。

□ キーアイデアは時間 t で資源制約を満足していなければ、t+n △でも満足していない、という点。

□ たとえばプログラム 20.4b で△=3 で試みる。マシンがロードは 1 つずつしか実行できないとすると、1 サイクルに 2 つロードを入れるスケジューリングは不可。

□ そこで片方のロードを 0 または 1 のサイクルに動かす。

□ もっと前や後に動かすことも考えられるが、mod が 1 になるサイクルに動かすことはできない。

3.2 レジスタ割当への影響

□ 図 20.7 のスケジュールでサイクル 0 にある $d \leftarrow fOPc$ を考える。この命令を後ろへ移すと f、c を定義する命令からの依存パスは長くなり、d を使う命令 $W[i] \leftarrow d$ までの依存パスは短くなる。

□ パス長が 0 以下になったらそのスケジュールは不可で、そのときは d を使う命令を後ろへ移してやる。

□ しかしそうすると、1 つのループ周回につき同じ値の「バージョン」を多く保持する必要があり (図 20.8 の f、f'、f'')、テンポラリの所要量が増え、レジスタ割り付けが失敗するかも。

□ つまり最適なループスケジューリングはレジスタ割り付けも同時に考慮する必要。

□ しかし最適アルゴリズムは実用でないかも→ここで説明しているアルゴリズムではまずスケジューリングし、次にレジスタ割り付けをやってみる。

3.3 最少周回数を見つける

□ まず周回数の下界を見つける。

- 資源に基づく見積り: 乗算器などは何サイクル所要か分かっているので、そのサイクル数をユニットの個数で割ると△の下界が求まる。6 個の乗算命令が必要で乗算器が 3 サイクル掛かり乗算器が 2 個あるなら、$\triangle >= 6 \cdot 3/2$。

- データ依存に基づく見積り: データ依存グラフで x_i が x_{i-1} に依存しているなら、その依存チェインの長さが△の下界。

□ たとえばプログラム 20.4b で、同時には 1 個の OP と 1 個のロード/ストアのみ発行可能とし、各命令は 1 サイクルで完了し、$i \leftarrow i+1$ や条件分岐のスケジューリングを考えないとすると。

- 演算の制約: 5 個の OP があるので △ >= 5。
- ロード/ストア制約: 4 個のロード/ストアがあるので △ >= 4。
- データ依存制約: グラフ 20.5a の $c_i \rightarrow d_i \rightarrow e_i \rightarrow c_{i+1}$。なので △ >= 3。

□ 次に、ループ本体の命令に何らかのヒューリスティックで優先順位をつける。

- 例: クリティカルなデータ依存サイクルに入っている命令を優先。
- 例: 貴重な資源を多数用いる命令を先に考える。

□ ここの例では $H = [c, d, e, a, b, f, j, g, h]$ の順で優先 (クリティカルなデータ依存サイクル中の命令、演算命令を優先)。

□ S : スケジュール済みの命令の集合、$SchedTime[h]$: スケジュールされた時刻 ($h \notin S$ ならば $none$)。

□ アルゴリズム 20.9 --- まだスケジュールされていない、優先度が最大の命令 h を S に入れる:

- まず、既に配置されている命令との依存関係、および資源制約を満たす最も 早いタイムスロットに入れようとする。
- ただし、候補として△サイクルぶん順に見てそのようなスロットが見出せないなら、それ以上見ても無駄 → 資源制約を無視して依存関係のみの基づいてタイムスロットを割り当てる。そのときは h を入れたことによって S が正しくなくなっているので、h の後続命令で依存制約を満たさなくなったものと、任意の命令で h と資源が衝突するものを S から取り除く。

□ このような試行錯誤は無限に続く可能性はあるが、通常はすぐに割り当てができるか、まったく割当ができないことが分かる。

- 試行数の上限 (予算 --- Budget) として、c × n 程度用意すればよい (c は 3 とか)。
- 上限までやって駄目なら、△を 1 ふやす。

□ (アルゴリズム 20.9 を読む)

□ ある変数 j の def-use 辺が△より長くなったら、j のインスタンス (コピー) が余分に必要で、それらを渡るための move 命令も必要

- Fig 20.8 の変数 a, b, f, j がそれに相当 (move 挿入アルゴリズムは省略)。

□ 資源の衝突チェックは長さ△の資源予約表を作れば定数時間で可能。

□ このアルゴリズムが最適なスケジュールを見つけるとは全く言えない。

- 可能な最少の△でのスケジューリングに失敗するかも。
- スケジューリングはできてもレジスタ割り付けに失敗するかも。
- しかし実用上はこのアルゴリズムは大変うまく行くとされている。

□ 適用例は Fig 20.10。

4 他の制御フロー

□ もしループ中に if-then-else があったら?

□ 方法 1: 両方の枝を計算して条件つき move 命令で正しい方の結果を転送 (条件つき move は多くの高性能のマシンにある)。

```
for i := 1 to N   for i :=  1 to N
  x := M[i]         x := M[i]
  if x > 0          u' := z * x
    u := z * x      u := A[i]
  else              if x > 0 move u := u'
    u := A[i]       s := s + u
  s := s + u
```

□ しかし、両方の枝のサイズが違って、頻繁に実行される側が軽いときは非効率。また副作用を持つ枝は無条件に実行してはいけない。

□ 方法 2: トレーススケジューリング --- 頻繁に実行される側 (トレース) だけに着目して効率のよいループを構成し、トレースから外れる分岐では非効率的になることを甘受。

5 コンパイラがスケジュールすべきか?

□ 多くのマシンは動的スケジューリングハードウェアを持つ。

- 「out-of-order 実行」 --- 複数のデコード済み命令をバッファに溜めておき、依存制約の満たされたものから順に実行開始。
- 1967 年の IBM 360/91 が最初だが、一般的になったのは 1990 年代半ば。
- 現在の高性能 CPU ではほとんどが動的スケジューリング機能を搭載。
- 利点も欠点もある→コンパイラ/ハードのどちらがスケジューリング、という問題は決着していない。

5.1 静的スケジューリングの利点

□ 動的→高価なハード資源を消費し、消費電力増/サイクルタイム増につながる。

□ 静的→将来の依存パスが長い命令を先に実行開始させられる (Ex 20.3)。

□ スケジューリング問題は NP 完全→コンパイル時の方が十分な時間が掛けられる。

5.2 動的スケジューリングの利点

□ キャッシュミスなどはコンパイル時には予測できない→動的が有利 (Fig 21.5)。

□ パイプラインスケジュール→多数のレジスタが必要→命令のフィールドでの制約→実行時のレジスタリネーミングならこの制約を逃れられる。

□ 最適な静的スケジューリングはパイプラインの状態を厳密に知らないとできない→実際には難しい。

□ 静的→命令セットが同じでも実装が違うとコンパイルし直しが必要。

6 分岐予測

□ 実数計算プログラムの多く (例: プログラム 20.4a) →基本ブロックが長く、命令も実行時間が長く (浮動少数点演算)、条件分岐も for の脱出などで行く方向がほぼ予測可能→前節のようなスケジューリング手法が有効。

□ コンパイラ、OS、ウィンドウシステム、ワープロ→基本ブロックが短く、命令は短時間で終わる整数演算、条件分岐の方向は予測しにくい→十分な速度で命令をフェッチし続けることが困難。

□ Fig 20.11 →比較/分岐/加算のパイプラインステージ。分岐が実行されるまで次の命令がフェッチできない (ストール)。

□ 4issue のスーパースカラーマシンで、分岐の後 3 サイクル待機だと、11 命令スロットが浪費されてしまう (3 × 4 - 分岐の 1)。

□ マシンによっては、分岐命令の次にある命令は常にフェッチ/デコード。分岐が起きた場合はそれは無駄になるが、起きなければその命令がすぐに実行できる。

□ 分岐命令があると分岐先の命令を常にフェッチするマシンも。

□ さらに、分岐側と非分岐側の両方を同時にフェッチするマシン (複雑!) も。

□ 最近のマシンでは、分岐の成否を予測。

- 静的予測: コンパイラが行う分岐予測。
- 動的予測: ハードウェアが分岐命令の行った方向を記憶して再利用。

6.1 静的分岐予測

□ コンパイラの予測結果→分岐命令に 1 ビットを設けてハードに伝える。

□ この 1 ビットを節約する (または命令セットの互換性を保つ) →上向き分岐は「取られる」、下向きは「取られない」と予測。

- 上向きはループのための分岐→ループ反復方向に取られる方が多い。
- 下向きは例外条件で飛び出す分岐→取られない方が多い。
- もし全ての分岐を「取られる」と予測するマシンだと、通常の実行の流れを飛び出し側にする→キャッシュ効率が悪くなる (21.2 節)。

□ このようなマシンの場合は、コンパイラは基本ブロックを並べ替えて「取られる」と予測される分岐は低い命令番地になるようにする。

□ 分岐方向を予測する簡単なヒューリスティック (すべて経験的には役に立つとされている):

- Pointer: ポインタの「等しい」判定 (p == null, p == q) → false
- Call: call を支配するコードへの分岐→ false
- Return: 戻り命令を支配するコードへの分岐→ false
- Loop: その分岐を含んでいるループのヘッダへの分岐→ true
- Loop': その分岐を後支配しないプリヘッダへの分岐→ true (Fig 18.7 のようなケースに対応)
- Guard: r が分岐の条件に現われ、行き先が分岐を後支配せずそこで r が行きている→ true

□ 1 つ以上にあてはまる場合もある→ヒューリスティックに優先順位、またはどれとどれが成立ならどうする、という対応表

6.2 コンパイラは分岐を予測すべきかどうか?

□ 静的な分岐予測は完璧だったとして 9%(C 言語) ないし 6%(Fortran) のミス。

- なぜ 100 は無理かというと、常に同じ方へ行くんだったらその条件分岐は無意味だから (!!)。
- なぜ Fortran の方がよいかというと、分岐がループ分岐で、またループ周回数が多いから。

□ プロフィールベースの予測→より完全に近い静的予測が可能。

- 上述の静的予測→ C 言語で 20%のミス→完璧の半分くらい。

□ ハードの分岐予測→分岐命令あたり 2 ビットで履歴を記憶→ C 言語で 11%のミス→プロフィールベースと同じくらい。

□ しかし、10%のミスだったらストールは多すぎ。ストールで 11 命令スロットの浪費、10 命令につき 1 回ストールだったら、マシンの能力は半分しか出せない。

□ だから、ハードとソフトの協調でもっとがんばらないと。20%のミスは「何もしないよりまし」程度。

7 Futher Reading

□ Hennessy and Petterson 1996 → 高性能 RISC アーキテクチャの多くの側面にアプローチした教科書。

□ Heinrich 1992 → MIPS R4000 のパイプライン制約 (Fig 20.1, 2 に利用)

□ 1970's の CISC → マイクロコード → 並列性、だがコンパイラには手が出なかった

□ Fisher 1981, 1983 → 軌跡スケジューリング (最初マイクロコード用、続いて VLIW マシン用)

□ Aiken and Nicolau 1988 → ループの 1 周ごとを独立にスケジュールしないでオーバラップさせる、というアイデア

□ マルチプロセサ用スケジューリング、反復モジュロスケジューリング、最短パススケジューリング等いろいろ。

□ 理論的には最短スケジューリングは整数計画法で可能だが指数時間。さらにレジスタ割り当ても難しい。

□ Ball and Larus 1993 → 本章の静的分岐予測。

□ Young and Smith 1994 → プロフィール方式。