不定サイクル演算を考慮した高位合成のスケジューリング

scheduling for high-level synthesis considering indefinite cycle operations

戸田勇希 Yuki Toda †
石浦菜岐佐 Nagisa Ishiura ‡
曽根 康介 Sone Kousuke ‡

1 はじめに

従来の高位合成のスケジューリング手法 [1] は, 演算の実行時間が一定であることを前提として, 各演算を実行する制御ステップ (サイクル) を一意に決定する. しかし, メモリアクセス演算や減算シフト型の除算などのように, サイクル数がオペランドの値によって実行時に変動する演算も存在する. 従来の高位合成では, このような演算のサイクル数は最大値, あるいは特定の値であると仮定してスケジューリングを行うが, 実行時のサイクル数が仮定と異なっていた場合には, 無駄な待ちが生じる可能性がある.

これに対し, 本稿では, 演算器の完了信号を元に, 各演算の実行タイミングを実行時の状況に応じて変更するスケジューリング, およびバインディングの手法を提案する.

2 従来法における不定サイクル演算の扱い

本稿では, 資源制約スケジューリング (与えられた資源制約下で実行サイクル数最小化を目指す) を対象とする.

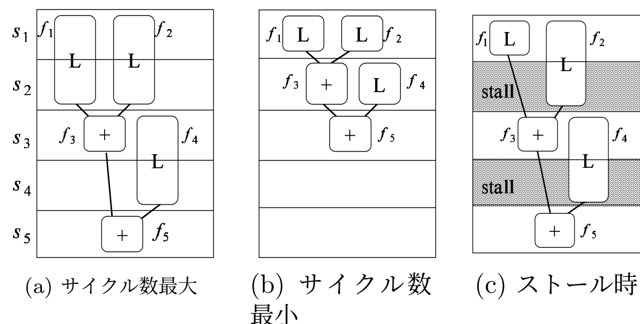
図 1(a) は, 加算器が 1 個, メモリアクセスユニットが 2 個使えるという制約下でスケジューリングを行った例である. ただし, 加算の遅延は 1 サイクルで固定だが, ロード (L) の遅延はオペランドによって (例えば, 連続するアクセスが同じラインに存在するかどうか等に依存して) 1 〜2 サイクルに変動するものとする. 従来の手法では, 例えばロード演算は最大の 2 サイクルを要するものとしてスケジューリングを行う. この場合, f_4 が 1 サイクルで完了したとしても, f_5 の実行を早めることはできない.

ロードが 1 サイクルで完了する確率が高い場合には, 図 1(b)(c) のように遅延を 1 サイクルとしてスケジューリングし, 1 サイクルで完了しなかった場合にはハードウェア全体をストールさせる方法も考えられる. これにより, f_4 の遅延に応じて f_5 の実行開始を調整することはできるが, f_1 と f_2 のいずれか一方だけが 1 サイクルで実行を完了しても f_4 の実行を繰り上げることはできない. 一般に, ある演算がハードウェアをストールさせると, その演算に依存しない全ての演算の実行が遅延させられる.

3 可変スケジューリング

本稿では, 各演算の実行に要するサイクル数はリストで与えられるものとする. 例えば, [2, 4, 6] は演算が 2, 4, または 6 サイクルを要することを表す. また, 演算 f_i を実行する演算器からその演算の完了信号 c_i が得られるものとする.

可変スケジューリングの結果は図 2 のように状態遷移グラフにより表現する. 各状態は, 静的スケジューリングの 1 サイクルに相当し, 枝はラベル付けされた条件成立時の遷移を表す. 開始状態 s_1 は静的スケジューリングの



(a) サイクル数最大　(b) サイクル数最小　(c) ストール時

図 1: サイクル数固定したスケジューリング

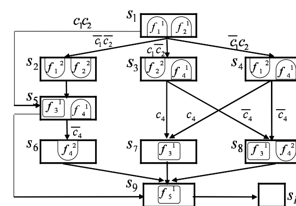


図 2: 可変スケジューリング

1 サイクル目に相当し, f_1 と f_2 の実行を開始する. 状態中の f_i^t は, f_i の t サイクル目の実行を表す. f_1 と f_2 がともに 1 サイクルで完了した場合は, $c_1 \cdot c_2$ の枝をたどって s_5 に遷移し, f_3 と f_4 の実行を開始する. s_1 で f_1 のみ 1 サイクルで完了した場合は, $c_1 \cdot \overline{c_2}$ の枝をたどって s_3 に遷移する. s_3 では, f_2 の 2 サイクル目を実行するとともに f_4 の実行を開始する. s_F は完了状態 (次の基本ブロックの DFG の先頭状態) である. 図 1 (c) のように f_1 のみが 1 サイクルで完了した場合は, $s_1 \rightarrow s_3 \rightarrow s_8 \rightarrow s_9 \rightarrow s_F$ という遷移を行い, 4 サイクルで処理を完了することができる.

4 スケジューリング・アルゴリズム

本稿では可変スケジューリングを求める一手法として, リストスケジューリングを拡張したアルゴリズムを提案する. アルゴリズムの概要を図 3 に示す. main では初期設定を行い, 再帰関数 schedule を呼び出してスケジューリングを行う. 2 行目の Status x はスケジューリング過程における各演算の実行状況を表すものである. x.exec は実行中の演算の集合, x.ready は実行可能な演算 (f の全ての親演算の実行が完了している) の集合, 関数 schedule は 状況 x から開始してスケジューリングを行い, 得られる状態遷移グラフの先頭の状態 (この場合は初期状態 s_1) を返す. 10 行目の State s はスケジューリングの 1 サイクルに相当する状態を表す. s.start は状態 s で実行を開始する演算の集合であり, s.cycle[f] は状態 s で演算 f の何サイクル目を実行しているかを示す. 13 行目でそれ以上実行する演算がなけば, 完了状態 s_F を返す. 14 〜 21 行目はリストスケジューリングにおける 1 サイクル分の処理に相当し, x.ready の演算のうち, その演算を実行で

† 関西学院大学 大学院理工学研究科 情報科学専攻
‡ 関西学院大学 理工学部 情報科学科

表 1: 可変スケジューリング・バインディング結果

プログラム	#op	#unit	静的 1	静的 2	可変スケジューリング			バインディング	
		(+, *, M)	#cy	#cy	#cy	#state	CPU (sec)	#st	CPU(sec)
matrix3.c	36	(3, 3, −)	13	14.9	10.9	246	0.3	246	52.8
ellip.c	40	(3, 3, 1)	18	15.4	12.5	327	1.1	327	199.6

サイクル数: +[1], *[2, 3, 4], M[1, 4]

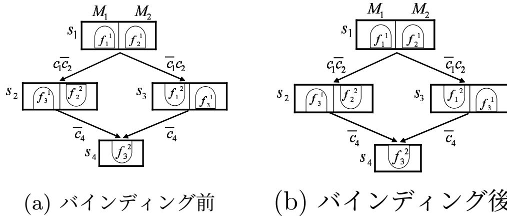


図 3: バインディングによる状態分割

きる演算器があるものをその状態にスケジューリングする. 重複する状態の生成を避けるため, 21 行目で同一判定を行い, すでに同一の状態があればその状態を返して完了する. 22 行目から 28 行目にかけて, 完了可能性がある演算の全ての組み合わせを生成し, それに対して schedule を再帰的に呼び出す.

5 バインディング

可変スケジューリングを用いた高位合成では, 従来のバインディングと異なり, 各演算 (値) に対して割り当てられる演算器 (レジスタ) は状態によって変化するため, 各々の割り当てで状態数増加を引き起こす可能性がある. 例えば, 図 2 のスケジューリングに対する演算器バインディングを考える. f_1, f_2, f_3 は 1〜2 サイクルのロード演算で, メモリアクセスユニットは 2 個 (M_1 と M_2) 使用できるとする. f_3 は, s_2 と s_3 で 異なるユニットに割り当てざるを得ない. s_2 と s_3 から $\overline{c_3}$ の条件で同じ s_4 に遷移するが, f_3 の 2 サイクル目を実行するユニットが異なるため, s_4 を 2 つの状態に分割する必要が生じる.

6 実験

本稿のスケジューリング, およびバインディング手法を Unix (Mac OS X) 上に Perl (5.8.6) で実装した. 結果を表 1 に示す. matrix3.c は 3 次正方行列の乗算, ellip.c はメモリアクセスを含むフィルタ演算である. #op は DFG 中の演算数, #unit は演算器数 (+ は加算器, * は乗算器, M はメモリアクセスユニット) である. 演算に必要なサイクル数は, 加算は [1], 乗算は [2, 3, 4] で, リスト中のサイクル数を等確率でとるものとし, メモリアクセスに必要なサイクル数は [1, 4] とした. メモリアクセスは, 前回と同じ行に対するアクセスは 1 サイクル, そうでない場合は 4 サイクルとした. 「静的 1」は各演算の最大サイクル数で静的スケジューリングを行ったもの. 「静的 2」は各演算の最小サイクル数で静的スケジューリングし, 実行時ストールを行うものであり, #cy は平均サイクル数である. 可変スケジューリング, およびバインディングの #state は状態遷移グラフの状態数, CPU は スケジューリングに要した計算時間である. 5 節で述べた状態分割の必要性に伴う状態増加は, レジスタの割当においても同様の問題が発生する. この問題を抑制するため, 本実験では, まず演算器バインディングを行った後に, 状態増加を引き起こさないようにレジスタバインディングを行う, というバインディングの実装を行った. サイクル数の差が大きい演算器を含む ellip では実行サイクル数削減の効果が大きい.

```
01: void main () {
02:     Status x;    /* 各演算の実行状況 */
03:     x.exec = φ;    /* 実行中の演算の集合 */
04:     x.ready =  実行可能な演算の集合;
05:     cycle[*] = 0;   /* 各演算のサイクル数初期化 */
06:     s_1 = schedule(x, cycle);
07: }
08:
09: State schedule (Status x, int cycle[]) {
10:     State s;    /* スケジューリングの 1 サイクルに相当 */
11:     s.start = φ
12:     s.cycle[*] = cycle[*];    /* f が s で何サイクル目か */
13:     if (x.ready == φ && x.exec == φ) return s_F;
14:     for (演算 f ∈ x.ready) {
15:        if (f を実行できる演算器が存在) {
16:              s.start に f を加える
17:              f を x.ready から x.exec に移す;
18:              s.cycle[f] = 0;
19:        }
20:     }
21:     for (f ∈ x.exec) s.cycle[f]++
22:     if (s と同じ状態 t が既に存在) return t;
23:     for (C ∈ 実行が終了し得る演算の全ての可能な組合せ) {
24:         x' = x;
25:         for (f ∈ C) f を x'.exec から削除;
26:         実行可能になった演算を x'.ready に追加
27:         s' = schedule(x', s.cycle);
28:         s から s' に 枝を張り C の条件をラベル付けする;
29:     }
30:     return s;
31: }
```

図 4: スケジューリング・アルゴリズム

7 むすび

本稿では可変スケジューリング, バインディング手法を提案した. 今後の課題としてスケジューリング・バインディング各々の段階における状態数削減や計算時間の短縮化, 合成後の回路規模や遅延評価などが挙げられる.

謝辞 本研究を進めるにあたり御助言・御討論を頂きました, 京都高度技術研究所の神原弘之氏, 名古屋大学の冨山宏之准教授に感謝します.

参考文献

[1] Daniel D. Gajski, Nikil D. Dutt, Allen C-H Wu, and Steve Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).