C-10

不定サイクル演算を考慮した高位合成の動的スケジューリング

Dynamic scheduling for high-level synthesis
considering indefinite cycle operations

戸田 勇希 †　　石浦 菜岐佐 †
Yuki Toda　　Nagisa Ishiura

1 はじめに

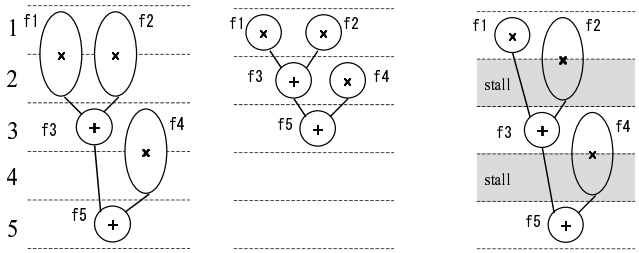
高位合成 [1] は, プログラミング言語などによるハードウェアの動作記述から論理合成可能なレジスタ転送レベル回路を自動設計する技術であり, ハードウェア設計を効率化する技術として注目を集めている. 高位合成の処理過程のうちスケジューリングと呼ばれる工程では, 各演算を実行するタイミング (および使用する演算器の種類) を決定する. 従来のスケジューリング手法は, 演算に要するサイクル数が一定であることを前提とし, 静的スケジューリングを行う [1]. しかし, メモリアクセス演算や減算シフト型の除算などのように, 実行サイクル数がオペランドの値や状況によって変動する演算も存在する. 従来の静的スケジューリングでは, 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していた (そのサイクル数で演算が完了しなければ回路全体をストールさせる) が, 無駄な待ちが生じる場合がある.

これに対し, 本稿では, 高位合成における動的スケジューリングを提案する. 本手法では, 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選択することにより, 不定サイクル演算を含むコードに対しても効率的なスケジューリングを実現することができる.

2 高位合成におけるスケジューリング

高位合成のスケジューリングは, データフローグラフ (DFG; 接点が演算, 枝が結合関係を表す) とデータパス情報 (各演算を実行する演算器とその個数および必要なサイクル数) を入力として, 各演算の実行するタイミングを決定する. 演算の依存関係や, 1 サイクルに利用可能な演算器に矛盾が生じない等の制約下で全体のサイクル数を最小にすることがスケジューリングの目標となる.

従来の静的なスケジューリングでは, 与えられた DFG に対し, DFG 中の各演算の実行を開始するクロックサイクルを決定していた. 図 1 に静的スケジューリングの例を示す. (a) は 1 サイクルの加算器が 1 つ, 2 サイクルの乗算器が 2 つという制約でスケジューリングを行った例である. ここで, 乗算のオペランドによっては 1 サイクルで演算が完了するという場合を考える. 例えば (a) のスケジューリングで, f4 が 1 サイクルで完了しても, f5 の実行を早めることはできない. これに対し, (b) のように乗算を 1 サイクルと仮定してスケジューリングを行い, 1 サイクルで完了しなかった場合には回路全体をストールさせるという方法が考えられる. 例えば (b) 中の乗算 f2, f4 が 2 サイクル必要とした場合, (c) のように回路をストールさせる. f4 のサイクル数に応じて f5 の開始サイクルを調整することはできるが, f1 と f2 のいずれか一方だけが 1 サイクルで完了してもそれをスケジューリングに反映させることはできない (f4 の実行開始を 2 サイクル目に繰り上げることはできない).



(a) サイクル数最大　(b) サイクル数最小　(c) ストール時

図 1: 静的スケジューリング

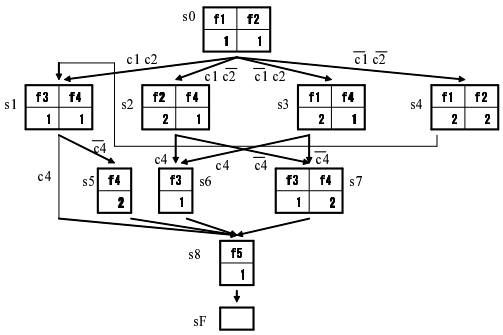


図 2: 動的スケジューリング

3 動的スケジューリング

本研究では, 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し, [1, 3, ∞] は, 1, もしくは 3 サイクル以上要することを表す. サイクル数固定の演算は [1] のように 1 要素のリストで表す. 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.

動的スケジューリングの結果は状態遷移グラフにより表現する. 図 1 の DFG に対する動的スケジューリングの結果を図 2 に示す. 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. 例えば, 図 2 において, 開始状態 s0 が静的スケジューリング

† 関西学院大学 理工学部 情報科学科

表 1: 各スケジューリング結果

プログラム	#op	#unit			静的	動的		
		+	*	M	#cy	#cy	#state	CPU (sec)
matrix3.c	27	3	3	–	11	10.6	145	0.36
	27	2	2	–	13	12.7	111	0.16
ellip.c	40	3	3	1	19	15.2	500	2.71
	40	2	2	1	20	17.4	306	0.82

サイクル数: +[1], *[2, 3, 4], M[1, 10]

の 1 サイクル目に相当し, f1 と f2 の実行を開始する. f1 と f2 がともに 1 サイクルで完了した場合は, $c1c2$ の枝を辿って s1 に遷移し, f3 と f4 の実行を開始する. s0 で f1 のみ 1 サイクルで完了した場合は, $c1\overline{c2}$ の枝を辿って s2 に遷移する. s2 では, f2 の 2 サイクル目を実行するとともに f4 の実行を開始する. sF は完了状態であり, 次の DFG の先頭状態を指す.

4 スケジューリング・アルゴリズム

本研究では動的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, 再起関数 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 は実行可能な演算の集合である. ready(x, f) は状況 x において, f が実行可能である (f の全ての親演算の実行が完了している) かを判断する関数である. 関数 schedule は 状況 x から開始して動的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を返す. s0 は状態遷移グラフの先頭の状態である. 10 行目の State s は静的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. 11 行目でそれ以上実行する演算がなけば, 完了状態 sF を返す. 12 〜 20 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 を再起的に呼び出す.

```
01: void main () {
02:     Status x;
03:     x.exec = φ;
04:     x.ready = { f | ready(x,f) == 1};
05:     cycle[*] = 0;
06:     s0 = schedule(x, cycle);
07: }
08:
09: State schedule (Status x, int cycle[]) {
10:     State s;
11:     if (x.ready == φ && x.exec == φ) return sF;
12:     s.cycle = cycle;
13:     for (演算 f ∈ x.ready) {
14:       if (f を実行できる演算器が存在) {
15:           s.start に f を加える;
16:           f を x.ready から x.exec に移す;
17:           s.cycle[f] = 0;
18:       }
19:     }
20:     for (f ∈ x.exec) s.cycle[f]++;
21:     if (s と同じ状態 t が既に存在) return t;
22:     for (C ∈ 実行が終了し得る演算の全ての可能な組合せ) {
23:       x' = x;
24:       for (f ∈ C) f を x'.exec から削除;
25:       x'.ready = x'.ready ∪ { f | ready(x',f) == 1};
26:       s' = schedule(x', s.cycle);
27:       s から s' に C に枝を張り対応する条件をラベル付けする;
28:     }
29:     return s;
30: }
```

図 3: スケジューリング・アルゴリズム

5 実験

前節のアルゴリズムに基づくスケジューリングプログラムを 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], メモリアクセス演算は [1, 10] とし, リスト中のサイクル数を等確率でとるものとした. 「静的」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で, #cy はサイクル数である. 動的スケジューリングの #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. サイクル数の差の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が, 状態数が 10 倍以上に増える. 計算時間は許容範囲内と考えられる.

6 むすび

本稿では動的スケジューリングを提案した. 今後は詳細な評価実験を行うとともに, 動的スケジューリングに対応したバインディングや制御回路の生成を行う予定である.

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助教授はじめ, HLS プロジェクトの諸氏に感謝します.

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).