社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2008－SLDM－137（24）
2008 / 11 / 19

不定サイクル演算を考慮した 高位合成の可変スケジューリング・バインディング

戸田 勇希† 石浦菜岐佐† 曽根 康介†

† 関西学院大学 理工学部
〒669-1337 兵庫県三田市学園 2-1

あらまし 本稿では, 高位合成における可変スケジューリングと, これに対応したバインディングの手法を提案する. 従来のスケジューリングでは, 全ての演算が固定のサイクル数で完了することを前提とし, 演算を実行するタイミングや各演算を実行する演算器を一意に決定していた. しかし, メモリアクセス演算などでは, 実行サイクル数がアドレスやメモリの状態により変化する. 従来法では, このような「不定サイクル演算」にも一定のサイクル数を仮定してスケジューリングを行っているが, 無駄な待ちが生じることがあり, その結果得られるハードウェアは必ずしも効率的とは言えない. 本稿では, 不定サイクル演算を含む動作記述に対しても効率的なスケジューリングを行える手法として, 可変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, 実行時の状況に応じて各演算の実行タイミングを選択することを許す. 可変スケジューリング, およびそのバインディングを高位合成システムに実装し, いくつかのベンチマークに対して評価実験を行った. その結果, 従来のスケジューリングと比較して状態数は増加するが, 総サイクル数は最大で 16 ～ 31% 削減することができた.

キーワード 高位合成, 動作合成, 可変スケジューリング, 不定サイクル演算

Variable Scheduling and Binding for High-Level Synthesis Considering Indefinite Cycle Operations

Yuki TODA†, Nagisa ISHIURA†, and Kousuke SONE†

† School of Science and Technology, Kwansei Gakuin University
2-1 Gakuen, Sanda, Hyogo, 669-1337, Japan

Abstract This article presents *variable* scheduling and binding for high-level synthesis. Conventional scheduling algorithms decide the operations' execution timing assuming that each operation takes a fixed number of cycles. However, on some operations such as memory accesses and serial multiplication/division, the number of cycles for the operation may vary depending on the values of operands or the states of the hardware. The variable scheduling enables efficient computation in the presence of such indefinite cycle operations where that timing of each operation execution is adaptively decided depending on the completion signals from the functional units. Experimental results show that the number of the execution cycles are reduced by 16 – 31%, although the number of states are increased as compared with the conventional scheduling algorithms.

Key words high-level synthesis, behavioral synthesis, variable scheduling, indefinite cycle operation

1. はじめに

高位合成 (動作合成) [1] は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, VLSI の設計を効率化する技術として実用化が進んでいる.

高位合成の処理過程のうち, スケジューリングでは各演算を実行するタイミングや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, 演算を開始するタイミングや, 演算を実行する演算器の種類を一意に決定する [1]. しかし, 演算のサイクル数は必ずしも一定ではなく, 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場合がある. 例えば, メモリアクセス演算や減算シフト型の除算は, アドレスやデータに依存してサイクル数が変動し得る. 従来のスケジューリングでは, こ

のような演算は, サイクル数を最大値, 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, 仮定したサイクル数以下で演算が完了しなければ回路全体をストール (停止) させていた. しかし, 実行時に演算の実行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ことになる.

これに対し, 本稿では, 高位合成における可変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. また, 本稿では可変スケジューリングに対応したバインディング手法を提案する. 従来のバインディングと異なり, 可変スケジューリングの結果に対しては各演算 (値) に対してそれを実行する演算器 (レジスタ) を必ずしも一意に決定できないため, これに適したバインディング手法を提案する.

上述のスケジューリング及びバインディングを高位合成システムに実装した. その結果, 状態数数が 18 倍に増加するが, 従来のスケジューリングと比して総サイクル数を 16 ～ 31% 削減できることができた.

以下, 第 2 節で不定サイクル演算, および従来法における不定サイクル演算の扱いについて述べ, 第 3 節で本論文で提案する可変スケジューリングとそのアルゴリズムの詳細を述べる. 第 4 節でこの可変スケジューリングに適したバインディング及びそのアルゴリズムを説明し, 第 5 節で本手法の実装, シミュレーションを行いその結果を検証する. 第 6 節で結論と今後の課題を述べる.

2. 不定サイクル演算とそのスケジューリング

2.1 サイクル数が不定の演算

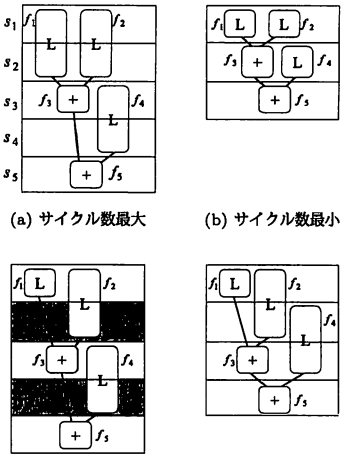
実行サイクル数が不定となる演算 (不定サイクル演算) の一例として, ロードやストアなどのメモリアクセス演算が挙げられる. メモリアクセスでは, キャッシュヒットするか否か (ヒットすれば 1 サイクル, しなければ 10 サイクルなど) や, バーストモード (最初のアクセスは常に 4 サイクル必要となるが, 以降の連続したアドレスに対するアクセスは 1 サイクルで完了する) などによりサイクル数が変動する. また, シリアル乗算器や減算シフト型の除算器でもオペランドの値によりサイクル数が変動する.

2.2 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 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, 必ず全体を 5 サイクルで完了できるが, 仮に実行時 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.

これに対し, (b) のようにロード演算を最小の 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. この場合, 最短で 3 サイクルで全演算を完了できるが, f_2, f_4 が 2 サイクル必要とした時には (c) のように回路をストールさせる必要があり, 全体で 5 サイクルを要する. しかし, もしあらかじめ f_2, f_4 に 2 サイクルを要することがわかっていれば, (d) のようにスケジューリングを行うことができ, 4 サイクルで計算を完了することができる. このように, 従来法では不定サイクル演算を含むコードに対しては必ずしも最適なスケジューリングを行うことはできない.



(a) サイクル数最大　(b) サイクル数最小

(c) ストール時　(d) f_2, f_4 のみ 2 サイクル

図 1: 従来法による不定サイクル演算のスケジューリング

3. 可変スケジューリング

3.1 可変スケジューリングとその表現

前節の問題を解決する手法として, 本稿では各演算を実行するタイミングおよび演算器の種類を実行時に決定する可変スケジューリングを提案する.

本手法は, スーパスケーラマイクロアーキテクチャ [2] と同様, 実行時に他の演算の完了状態に依存して次に実行する演算を変更できる. しかし, これをリザベーションステーションやリオーダバッファを用いた複雑なハードウェアではなく, あらかじめ計算した状態遷移グラフに基づく制御により実現する.

本手法では, 各演算の実行に要するサイクル数はリストで与るものとする. 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.

従来, スケジューリングの結果は図 1 のように表現されてきたが, 本稿ではこれを一般化し, 状態遷移グラフにより表現する. 例えば, 図 2 は 図 1(a) の スケジューリング を状態遷移グラフで表現したものである. 各状態は, 従来のスケジューリングの 1 サイクルに相当する (S_F は終了を表す仮想的な状態である).

図 1 の DFG に対する可変スケジューリングの結果を図 3 に示す. 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 3 において, 開始状態 s_1 が従来のスケジューリングの 1 サイクル目に相当し, f_1 と f_2 の実行を開始する. f_1 と f_2 がともに 1 サイクルで完了した場合は, c_1c_2 の枝を辿って s_2 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, $c_1\overline{c_2}$ の枝を辿って 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 は完了状態であり, 次の DFG の先頭状態を指す.

これにより, 図 1 (a), (b), (d), いずれの場合 (その他の遅延の組み合わせの場合) においても最適な実行スケジューリングを得ることができる.

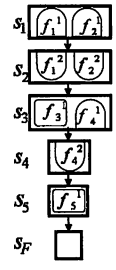


図 2: 図 1(a) のスケジューリングの状態遷移グラフ

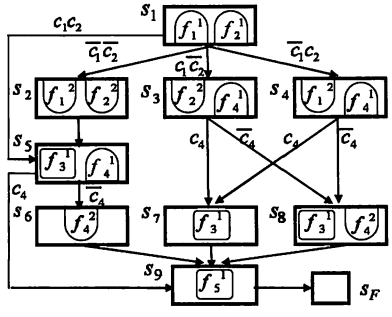


図 3: 可変スケジューリング

3.2 可変スケジューリングの定式化

F を演算の集合, M を演算器の種類の集合とする. 演算 $f \in F$ に対し, $P(f)$ を演算 f が依存する演算の集合, $D(f)$ を f の実行に必要なサイクル数の集合とする. $M(f)$ を f を実行できる演算器の種類の集合とし, $m(s,f)$ は状態 s で f を実行する演算器の種類とする. また, $m \in M$ に対し, $n(m)$ を m の個数とする. S を状態 (実行サイクル) の集合とする. S は初期状態 s_1, および終了状態 s_F を含むものとする. $I(f) \in S$ は f の実行を開始する状態の集合, $\delta(s,Z)$ は状態 s で実行が完了する演算の集合が Z に等しいときに遷移する次状態を表す. $x(s,f)$ は状態 s で f の何サイクル目を実行しているかを表し, 以下のように定義される. ただし, $x(s,f) = \perp$ は f が s で実行されていないことを表す.

- $x(s,f) = 1 \Leftrightarrow s \in I(f)$.
- $s' = \delta(s,Z)$ に対し, $x(s',f)$

$$= \begin{cases} x(s,f)+1 & (x(s,f) \neq \perp \wedge f \notin Z \text{ のとき}). \\ \perp & (\text{上記以外のとき}). \end{cases}$$

- それ以外の s と f に対して $x(s,f) = \perp$.

ただし, $s' = \delta(s,Z) \wedge x(s,f) \neq \perp$ のときに $s' \in I(f)$ であってはならない. 即ち, f の実行を開始する状態から遷移できる状態の中に, f を開始する状態があってはならない. また, (s,s',Z_1,Z_2) s.t. $\delta(s,Z_1) = \delta(s',Z_2)$ であるときには,

$\forall f \in F : x(s,f) \vee (x(s',f) \neq \perp \vee f \in Z_1) \wedge (x(s',f) \neq \perp \wedge f \in Z_2) = x(s',f)$

でなければならない. つまり, 同一の状態へ遷移する異なる 2 つの状態があり, f が次状態でも実行中であれば, そのサイクル数が一致していなければならない. 状態遷移においては, 遷移時完了する演算のサイクル数が $D(f)$ に含まれていなければならない.

$$s' = \delta(s,Z) \Rightarrow \forall f \in Z : x(s,f) \in D(f).$$

ただし, 逆は必ずしも成立する必要はない. 即ち, $x(s',f) \in D(f)$ であっても, $s = \delta(s,Z), f \in Z$ なる Z が存在する必要はない.

$A(s)$ は状態 s の実行開始時点に実行が終了している演算の集合であり, 次のように定義する.

- $A(s_0) = \phi$.
- $A(\delta(s,Z)) = A(s) \cup Z$.

ただし, $\delta(s_1,Z_1) = \delta(s_2,Z_2)$ ならば $A(s_1) \cup Z_1 = A(s_2) \cup Z_2$ でなければならない. これは同一の状態へ遷移する異なる 2 つの状態があれば. 完了した演算が一致していなければならないことを意味する.

可変スケジューリングは以下の制約を満たさなければならない.

(1) 依存制約

$$\forall f \in F, \forall s \in I(f) : P(f) \subseteq A(s).$$

これは, f の開始時点で, f が依存している全ての演算が完了していなければならないことを表す.

(2) 資源制約

$F(s,m) = \{f \in F \mid x(s,f) \neq \perp, m(s,f) = m\}$. とすると,

$$\forall s \in S, \forall m \in M : |F(s,m)| \leq n(m).$$

これは, 各状態で使用される演算器の数が使用可能な演算器数を超えないことを規定する.

(3) 終了制約

$$\forall q \in Q \ : \ A(s_e(q)) = F.$$

ただし,

$$Q = \{s_0 s_1 s_2 ... s_k \in S^+ \mid \exists Z \ \ s_{i+1} = \delta(s_i, Z) \ \ (\text{for } i = 1,2...,k-1) \ \ \forall Z \ \ \delta(s_k,Z) = \perp\}.$$

これは, 最終的に全演算が実行完了状態になることを規定する.

以上の制約から, 全パスのサイクル数の合計を最小化するこ

とを目標とする.

$$\text{minimize} \sum_{q \in Q} |q|.$$

3.3 可変スケジューリングのアルゴリズム

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 4 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) の集合を表す. 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る. 13 行目でそれ以上実行する演算がなければ, 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, 22 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. 23 行目から 29 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.

```
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:     s1 = 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:     x(s, *) = cycle[*];    /* f が s で何サイクル目か */
13:     if (x.ready == φ && x.exec == φ) return sF;
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:           x(s,f) = 0;
19:       }
20:     }
21:      for (f ∈ x.exec) x(s,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', x(s,*));
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: 可変スケジューリングのアルゴリズム

4. 可変スケジューリングに対応したバインディング

4.1 従来のバインディングとの相違

従来のスケジューリング結果に対するバインディングでは, 演算に対する演算器割り当て, および値に対する記憶要素の割り当ては, 状態に依存せず決定することができた. しかし, 可変スケジューリングに対するバインディングでは, これらの割り当ては状態に依存して変化することがある.

例えば可変スケジューリングによって図 5(a) のような状態遷移グラフが得られたとする. 演算 f_1, f_2, f_3 は演算器 M_1, M_2 のいずれかで実行できるとする. 図 5(b) は図 5(a) の各演算に対し演算器割り当てを行ったものである. 状態 s_1 で演算 f_1, f_2 をそれぞれ M_1, M_2 で実行するとする. 状態 s_2 では, f_1 は完了し f_2 の実行を継続するため, f_2 は M_1 で実行しなければならない. これに対し, s_3 では f_3 は空いている M_2 で実行しなければならない. 即ち演算 f_2 をどの演算器で実行するかは状態によって変化する. (注1)

図 5(c) は図 5(b) に対しレジスタバインディングを行った結果である. 状態 s_2 では, レジスタ R_3 と R_4 は f_1 の入力値を保持するため, f_3 の入力値は別のレジスタ R_1, R_2 に割り当てる. これに対し, s_3 では, R_1, R_2 は f_2 のために使用しているので, 別のレジスタ R_2, R_4 に割り当てる. このように, 値のレジスタへの割り当ても状態に依存して変化することになる. (注2)

演算の結果を書き込むレジスタは, 遷移先の状態でその結果を読み出すレジスタに等しい. 即ち, 演算の結果を書き込むレジスタは, 現状態と完了信号の組み合わせによって決まる.

本手法により合成される回路は基本的に従来と同じである. 合成されたハードウェアの制御回路は状態遷移に基づいて, 演算器の入力となる値の選択, レジスタの入力の選択, 書き込みの有無を制御する信号を出す. 従来法と異なる点は, 完了信号が制御回路の入力となっていることである.

4.2 状態の分割

可変スケジューリングに対応するバインディングでは, スケジューリングにより得られた状態遷移グラフに対して状態の分割が必要になる場合がある.

(1) 演算器割り当てに伴う状態数分割

図 6(a) は図 5(b) に状態 s_4 を追加したものである. 状態 s_2 と s_3 で演算 f_3 は異なる演算器で実行するため, 両方の状態から s_4 に遷移を許すと, 状態 s_4 で矛盾が生じてしまう. そのため, 図 6(b) のように状態 s_4 を 2 つに分割しなければならない.

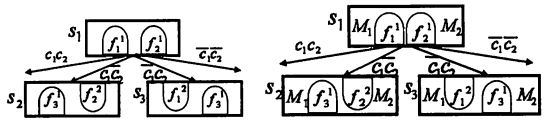
(2) 記憶要素割り当てに伴う状態数分割

(注1): もう 1 つの演算器 M_3 が利用できれば, f_3 に M_3 を割り当てることにより, 割り当てを状態に依存しないようにできる.

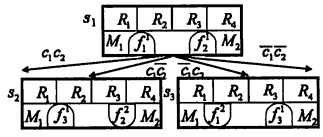
(注2): レジスタの数が十分にあれば, f_3 の入力に例えば R_5, R_6 を用いることにより, 割り当てを状態に依存しないようにできる.

図 7(a) のような状態遷移グラフを考える. f_3 の入力値は状態 s_2 では R_1, R_2 に, 状態 s_3 では R_3, R_4 に割り当てているため, 同じ状態 s_4 への遷移はできない. そのため, 図 7(b) のように状態 s_4 を 2 つに分割しなければならない.

このような状態分割の増加は, 避けられないものもあるが, 演算器やレジスタの割り当てによっては避けられるものもある. 例えば図 7 において, s_2 で演算 f_3 の入力値に対して R_3, R_4 を割り当てればこの分割は避けることができる. このように, バインディングにおいては従来の指標に加え, 状態数の増加を抑制することも重要な目標となる.

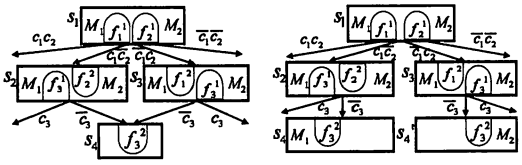


(a) スケジューリング結果　(b) 演算器バインディング



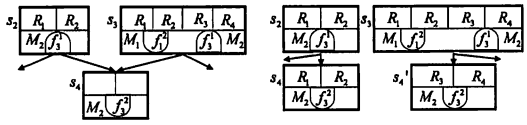
(c) レジスタバインディング

図 5: バインディング



(a) 分割前　(b) 分割後

図 6: 演算器バインディングによる状態分割



(a) 分割前　(b) 分割後

図 7: レジスタバインディングによる状態分割

4.3 可変バインディングの定式化

各演算の入出力となる値の集合を V , レジスタの集合を R とする. また, 入力ポートの集合を PI, 出力ポートの集合を PO とする. 演算器の種類 $m \in M$ に対し, 種類 m の演算器の集合を $U(m)$ とする, $pi(u,k)$ を演算器 u の k 番目の入力ポート, $po(u,k)$ を演算器 u の k 番目の出力ポートとする. $vi(f,k)$ は演算 f の k 番目の入力値, $vo(f,k)$ は演算 f の k 番目の出力値である. $nvi(f)$ は演算 f の入力ポート数, $nvo(f)$ は演算 f の出力ポート数とする.

バインディングでは, 状態 s で f を実行する演算器 $u(s,f)$ (ただし $u(s,f) \in U(m(f))$) と, 状態 s で 参照する v を格納するレジスタ $r(s,v)$ を決定する. ただし, 以下の条件を満たさなければならない.

- $\forall s \in S, \forall f_1, f_2 \in V \ : \ u(s,f_1) \neq u(s,f_2)$.
- $\forall s \in S, \forall v_1, v_2 \in V \ : \ r(s,v_1) \neq r(s,v_2)$.
- $s' = \delta(s,Z), x(s,f) \neq \perp \ \wedge \ x(s',f) \neq \perp$
  $\Rightarrow \begin{cases} u(s,f) = u(s',f), \\ 1 \leq k \leq nvi(f), \ r(s,vi(f,k)) = r(s',vi(f,k)). \end{cases}$

3 番目の制約は実行に複数サイクルを要する演算に対する制約で, 実行開始時点で割り当てられた演算器を途中で変更してはならないことを表す. 従来のバインディングと同様, 以上の条件から結線の数を最小にすることを可変バインディングの目標とする.

$$\begin{aligned} &\text{minimize}|C| \\ &C = \{(r(s,vi(f,k)), pi(u(s,f),k)), \\ &(po(u(s,f),l), r(s,vo(f,l))) \ | \\ &1 \leq k \leq nvi(f), 1 \leq l \leq nvo(f)\} \end{aligned}$$

4.4 バインディングのアルゴリズム

従来法では各状態における演算器割り当て, およびレジスタ割り当てを同時に行う. しかし, 可変スケジューリングに対するバインディングで同時に演算器・レジスタ割り当てを行うと状態数が急激に増大する [3]. そこで本稿では初めに全ての演算に対して演算器割り当てを行い, その後にレジスタ割り当てを行う方法を提案する.

本手法におけるバインディングのアルゴリズムの概要を図 10 に示す. 最初の main 関数では, 可変スケジューリングによって得られた開始状態 s0 を引数として, 演算器割り当てを行う再帰関数 fu_bind 関数, レジスタ割り当てを行う reg_bind 関数を呼び出す. fu_bind 関数では, 8 行目で受け取った状態が最終状態であればバインディングを完了する. そうでない場合, 10 ～ 12 行目で引数で与えられた状態で実行を開始する演算に対して利用可能な演算器の割り当てを行い, 13 ～ 28 行目は s の次状態 s' に対して再帰的に fu_bind 関数を呼び出す. 14 行目で s' がバインディング済みでないかを調べ, もし既にバインディングされていれば, 15 行目でそのバインディングが現状態 s のバインディングと矛盾が生じていないかを確認する. もし矛盾が生じていれば, 16 ～ 21 行目で s' を分割して new_s' を作成し, 22 行目でこれに対して fu_bind を呼び出す. 次状態がバインディング済みでなければ, s' に対して fu_bind を再帰的に呼び出す (25 ～ 26 行目). reg_bind 関数は, 状態 s0 を初期状態とする状態遷移グラフ (演算器割当て済み) を受け取り, レジスタ割り当てを行う. 34 行目で 各値に対してその値が生存する状態リストを作成し, 35 行目で被覆問題を解いて (貪欲法による), レジスタ割り当てを行う.

5. 実験と評価

5.1 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

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	従来 1	従来 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	0.31	246	0.34
ellip.c	40	(3, 3, 1)	18	15.4	12.5	327	1.13	327	0.98

サイクル数: +[1], *[2, 3, 4], M[1, 4]

```
01: int main () {
02:     fu_bind(s0);
03:     reg_bind(s0);
04: }
05:
06: void fu_bind (State s) {
07:     if (s が 最終状態 sF) {
08:         return;
09:     }
10:     for (f ∈ s.start 中の演算){
11:         f に利用可能な演算器の番号を割り当て;
12:     }
13:     for (s' ∈ s の次状態) {
14:         if (s' が既にバインディングされている) {
15:             if (s と s' のバインディングに矛盾が生じる) {
16:                 Status new_s';
17:                 new_s' に s' のスケジューリング情報を代入;
18:                 s から new_s' への枝を張り,
19:                 遷移条件は s から s' への枝の条件を複写;
20:                 s から s' への枝を削除;
21:                 s のうち次状態に引き継ぐ情報を new_s' へ複写;
22:                 fu_bind(new_s');
23:             }
24:         } else {
25:             s のうち次状態に引き継ぐ情報を s' へ複写;
26:             fu_bind(s');
27:         }
28:     }
29:
30:     return;
31: }
32:
33: void reg_bind (State s) {
34:     各値に対してその値が生存する状態リストを作成;
35:     被覆問題を解き (貪欲法による), レジスタ割り当てを行う;
36: }
```

図 8: バインディングのアルゴリズム

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. ただし, 1 行 256 ワード, 1 ワードあたり 32 bit である. 「従来 1」は各演算の最大サイクル数で従来手法のスケジューリングを行ったもの. 「従来 2」は各演算の最小サイクル数で従来手法のスケジューリングを行い, 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. 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.

6. むすび

本稿では可変スケジューリングとそれに対するバインディング手法を提案した.

可変スケジューリングおよびバインディングでは, 状態数増加をいかに小さく抑制するかが重要な課題となる. サイクル数削減に貢献しない状態, もしくは, 平均サイクル数を大きく増加させない遷移を削除することにより, 状態数を削減する方法も考えられる. また, 総サイクル数と状態数のトレードオフを考えることも重要になると考える.

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. また, 関西学院大学の入谷賢孝氏をはじめ, 石浦研究室の諸氏に感謝します.

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).

[2] 中森 章: マイクロプロセッサ・アーキテクチャ入門, CQ 出版 (2004).

[3] 戸田 勇希: "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," 関西学院大学 理工学部 情報科学科 2008 年度 卒業論文 (Mar. 2008).