計算機アーキテクチャ　98－9
設　計　自　動　化　65－9
（1993. 1. 21）

メモリ割り付け制約を考慮したスケジューリング手法

池田充郎, 宮崎敏明
NTT LSI 研究所
〒243-01 神奈川県厚木市森の里若宮 3-1
e-mail: ike@nttica.ntt.jp, tana@nttica.ntt.jp

あらまし　本稿では、動作記述中の変数の、メモリあるいはレジスタファイルへの割り付けを考慮したスケジューリング手法を提案する。従来、上位合成研究においては、動作記述中の変数はすべてレジスタに割り当てることを前提とするアプローチがほとんどであり、メモリあるいはレジスタファイルの割り付け問題はあまり議論されていなかった。しかし、実際の設計ではすべての変数をレジスタに割り当てるという前提は現実的ではなく、通常はレジスタに加えてメモリあるいはレジスタファイルなどが必要になる。メモリあるいはレジスタファイルへの割り付け問題は、最終的な回路のパフォーマンスを左右するものであり、今後の上位合成研究において避けて通れない問題である。ここでは、回路のパフォーマンスを決定するスケジューリングの段階において、メモリあるいはレジスタファイルの割り付けを考慮した手法を提案する。従来の演算器数などの制約に加えて、レジスタ数の制約および、メモリあるいはレジスタファイルのアクセス速度やポート数の制約を満たし、かつ、高速な処理を実現するスケジューリング法を示す。

和文キーワード　上位合成, スケジューリング, データフローグラフ, メモリ割り付け

A Scheduling Method considering Memory Allocation

Mitsuo Ikeda, Toshiaki Miyazaki
NTT LSI Laboratories
3-1, Morinosato Wakamiya, Atsugi-Shi Kanagawa Pref., 243-01, JAPAN

Abstract　This paper presents a scheduling method addressing the memory allocation of variables in behavioral description. Up to now, in high-level synthesis, many approaches assume that all variables are to be allocated to registers only. However, the memory(or register-file) allocation scheme has a great influence on the performance of the circuit. Therefore, good algorithms to allocate variables to memory or register file, are indispensable to realize practical high-level synthesis system, especially with the progress of the integration and the increase in circuit size. In this paper, we propose a scheduling method that considers the allocation of variables to both registers and memories. The constraints include the number of registers, the access speed and the number of memory- or register-file-ports. The algorithm we present realizes nearly optimal schedule satisfying the constraints. Benchmark data show the effectiveness of this method.

英文 key words　high-level synthesis, scheduling, data-flow graph, memory allocation

1 はじめに

近年のLSIの高集積化, 多機能化にともない、LSIの設計には多大な労力が必要となってきている。設計のTAT短縮化のために、抽象度の高い動作記述から回路を合成する上位合成技術の実用化が強く望まれている。

従来の上位合成研究では、動作記述中の変数がもつ値をすべてレジスタに保持するという前提のアプローチがほとんどであった。しかし、大規模回路の合成においてはすべての変数をレジスタに割り付けることは現実的ではない。実際の設計では、独立したレジスタの他にレジスタファイルや内部メモリなどの記憶素子が混在するのが通常であり、そのレジスタファイルやメモリの使用法が最終的なパフォーマンスを左右する問題になっている。

上位合成の研究においても、最近、変数のメモリあるいはレジスタファイルへの割り付け問題を扱うものがいくつか見られるようになってきた[3][4]。これらは、演算等のリソース割り付けの際に、変数をレジスタファイル等へ割り付けるものである。しかしながら、最終的なパフォーマンスを考えると、その前工程であるスケジューリングの段階から、メモリ等の割り付け問題を考慮する必要がある。従来、このようなアプローチをとる研究はなかった。

本稿では、動作記述中の変数のメモリ割り付けを考慮したRMAスケジューリング法(Register-and-Memory-Allocation-constrained Scheduling)を提案する。このスケジューリング法は、演算器数やレジスタ数などファシリティの制約に加えて、メモリあるいはレジスタファイルのアクセス速度やポート数の制約を導入し、できるだけ高速な処理の実現をめざすものである。

1.1 関連する研究

これまでの上位合成研究の中でメモリ割り付け, レジスタのグループ化を扱うものがいくつか見られる[2][3][4]。そのほとんどはスケジューリングの結果を受け、演算等のファシリティ割り付けと同時あるいはその前後の工程として、変数をメモリ等へ割り付けるものである。

しかし、メモリへのアクセスには、演算実行と比べて時間がかかり、また、ポート, バスを同時に使用できないなどの制約がある。そのため、スケジューリングの結果を受けた後に、割り付けを行なった場合、スケジューリングの結果そのままではなく、時間的にくずしたり、引き延ばしたりする必要が出てくる。すなわち、再スケジューリングが必要になる可能性がある。

一方、スケジューリング問題については、これまで多くの手法が提案されてきた[1]。その中で、スケジューリングの段階で、レジスタ数をできるだけ抑える手法としてはJITスケジューリング法[9], 演算のクラスタリングによりレジスタ数を抑える手法[7]などが挙げられる。最近では、演算器やレジスタなどのリソース割り付けをスケジューリングと同時に行なう手法[5][6]が提案されている。しかし、いずれのスケジューリング手法も変数をすべてレジスタに割り付けることを前提としたものであり、レジスタ以外のファシリティへの割り付けは考慮されていない。現在のベンチマーク程度の規模では、変数をすべてレジスタに割り付けたとしてもさほど問題にはならないが、より大規模な入力記述に対しては、レジスタを限られた資源と仮定する必要が出てくる。

1.2 本手法の特徴

本稿では、以上の問題点を解決する新たなスケジューリング法—RMAスケジューリング法を提案する。以下に、RMAスケジューリング法の特徴を簡単に述べる。

- レジスタ数, メモリ割り付け問題などリソース制約をスケジューリング時から導入することができる。
- 本手法の基本アルゴリズムはListスケジューリング法[1]を拡張したもので、性能の下限がListスケジューリング法の性能と等しいことが示される。
- 従来の整数線形計画法を用いたスケジューリング法などに比較して高速で同等の最適化能力を持つ。

以下、第2章では、本稿におけるスケジューリング問題のモデルを示す。第3章において、RMAスケジューリングアルゴリズムを説明し、ベンチマークデータに対し実験した結果を第4章に示す。最後にまとめを第5章で述べる。

2 スケジューリング問題のモデル

通常、レジスタやメモリなどの記憶素子において、アクセス速度とコストのトレードオフが存在する。例えば、より高速だがコストの高い単独のレジスタと、より低速だが低コストなメモリとが存在した場合などである。すべてをレジスタに割り付ければ、もっともパフォーマンスのよい合成が可能であるが、レジスタを限られた資源と仮定すれば、メモリなどのより安価なものへ割り付けるべきである。しかし、この割り付けを演算等のリソース割り付けの段階で行なうとすると、既に決定されたスケジュールを時間的に引き延ばさなければならない場合が出てくる。その場合、変更されたスケジューリング結果はもはや最適なパフォーマンスを保証することはできない。最終的なパフォーマンスの最適化を考えた場合、スケジューリングの段階でメモリあるいはレジスタへの割り付けを考慮する必要がある。

今回のスケジューリング問題のモデルとしては、単独のレジスタの個数を制約条件として与えるものとし、そのレジスタ数を越える場合は、変数をメモリないしレジスタファイルに割り当てるというアプローチをとる。以下に、スケジューリング問題のモデルを示す。

スケジューリング問題のモデル

- **最小化目的関数:**
  - 与えられた記述を実現するために必要な制御ステップ数
- **演算器などの制約条件:**
  - 演算器の種類と種類毎の個数および演算器が必要とする制御ステップ数
  - レジスタ数
- **メモリから来る制約条件:**

- データの読み書きに必要な時間
  (制御ステップ数) の制約
- ポート数の制約

ここで、“制御ステップ (C-Step)” とは、制御の状態遷移の一状態を表すもので、動作を実現するのに必要な制御ステップの数は、回路の処理速度に直結する。“レジスタ数” とは、各制御ステップで必要となるレジスタ数の最大値である。演算の入出力となる変数データは、その演算の直前, 直後で必ずレジスタに入れるものとする。今回のスケジューリングのモデルでは演算等のリソース割り付けは同時には行なわず、メモリなどの割り付けにともなう時間的制約のみをスケジューリングの制約条件として導入する方針をとる。

3 RMA スケジューリング手法

3.1 アルゴリズム

前述のスケジューリング問題に対する提案アルゴリズムを示す。アルゴリズムの概略は、次のようになる。最初、すべての変数にレジスタを使うと仮定し、レジスタをできる限り制約内に抑えるスケジューリングを行なう。その結果、レジスタ数が制約を越えた制御ステップについて保持しなければならない変数の中から、データのアクセスにもっとも時間的余裕のあるものを選び、メモリに割り当てる。メモリのアクセスからくる制約を満たしながら、再度、スケジューリングを行なう。この操作をレジスタ数の制約を満たすまで繰り返す。すなわち、この RMA スケジューリングアルゴリズム全体は、変数をメモリに割り当てながら、レジスタ数の制約が満たされるまでスケジューリングを繰り返して行なう。繰り返して行なうスケジューリング 1 回分は、List スケジューリング法 [1] を拡張したものである。以下、これを Look-ahead List スケジューリング法と呼ぶことにする。

以下、アルゴリズムを簡単な例題を用いて詳細に述べる。図 1, 図 2は “Diffeq”[8] と呼ばれる例題に対する動作記述およびそのデータフローグラフ (DFG) である。 ここで

while $(x < a)$ **repeat:**
$xl = x + dx;$
$ul = u - (3 * x * u * dx) - (3 * y * dx);$
$yl = y + (u * dx);$
$x = xl; u = ul; y = yl;$
end;

図 1: 動作記述の例

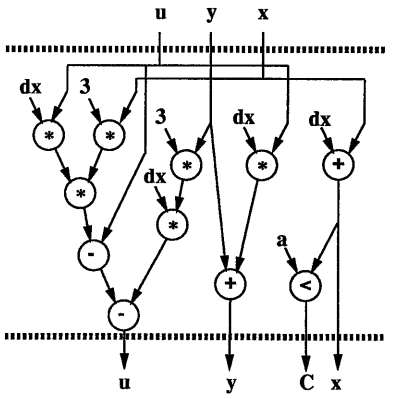


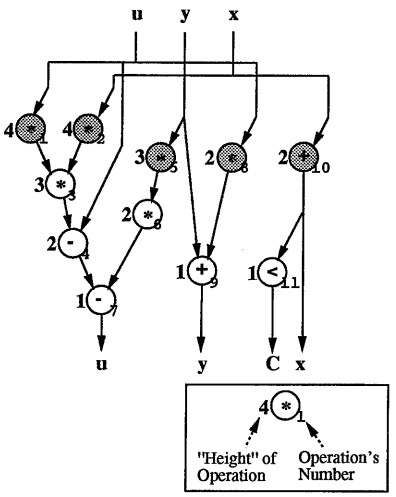
図 2: データフローグラフ

は、この動作記述に対して、図 3に示す条件下でのスケジューリング問題を解くことにする。

最小化目的関数: 制御ステップ数
制約条件: 乗算器数 ≤ 1
加算器数 ≤ 1
減算器数 ≤ 1
比較演算器数 ≤ 1
レジスタ数 ≤ 4
メモリアクセス時間 ≥ 2 制御ステップ
メモリ書き込みポート数 ≤ 1
メモリ読み出しポート数 ≤ 1

図 3: スケジューリング問題

第 1 段階として、List スケジューリング法を拡張した Look-ahead List スケジューリングを行なう。List スケジューリングは、演算のもつデータ依存関係から、対象とする制御ステップに割り当てることができる演算をある優先基準によってリスト形式に並べ、演算器制約を満たす範囲で優先度の高い順に演算を割り当てるものである。List スケジューリングにおいては、演算をリストの中からひとつずつ割り当てていくが、Look-ahead List スケジューリング法では、対象とする制御ステップに割り当てる複数の演算を同時に決定する。そのために、その制御ステップに割り当てることのできる演算の組合せをすべて求め、その中から、レジスタ数等の制約を満たし、かつ定めた優先基準による優先度の最も高い組合せを選択する。優先基準としては、ある演算を実行してから、データフローグラフ上の処理が終了するまでの最小制御ステップ数 (以下、演算の “高さ” と呼ぶ) をとる。図 4の上のデータフローグラフにおいて、演算ノードの左の数字がその演算の “高さ” を表している。なお、ノードの右下の数字は演算の番号を示す。



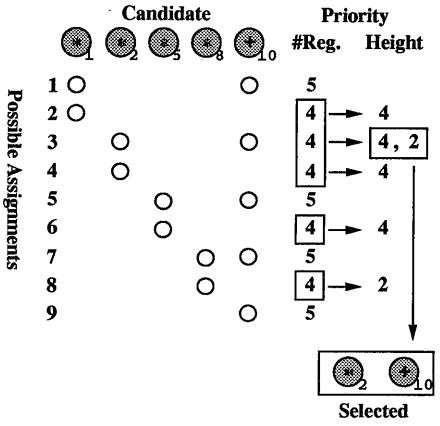


図 4: 割り当ての決定法

先ほどの "Diffeq" のスケジューリング問題に対して、第 1 制御ステップに割り当てられることのできる演算は図 4において黒いノードで示した 5 つの演算である。この 5 つの演算の組合せのうち、演算器制約から、可能な割り当ては、9 通り存在する。そのうち、レジスタ数の制約を満たす組合せは 5 通りであり、その中から、演算の "高さ" の優先度が最も高い組合せを選択する。この場合、乗算 2 と加算 10 の組合せ (図 4の下の図においては上から 3 番目の組合せ) が選ばれ、この 2 つの演算が第 1 制御ステップに割り当てられる。この操作を、次の制御ステップについて行ない、以下同様に繰り返す。Look-ahead List スケジューリングを 1 回行なった結果が、図 5である。図 5の制御ステップの境界線 (以下、ステップラインと呼ぶ) の右に記した数字は、各ステップラインで必要となるレジスタ数を表している。例題では、各ステップラインのレジスタ数の最大値が 5 となったが、これは第 2 制御ステップにおいてレジスタ制約を満たす割り当てが存在しなかったためである。なお、この例題では、演算器各 1 個を用いて 7 制御ステップで処理を行ない、かつ、変数をすべてレジスタに割り当てる場合、最低 5 個のレジスタが必要である。

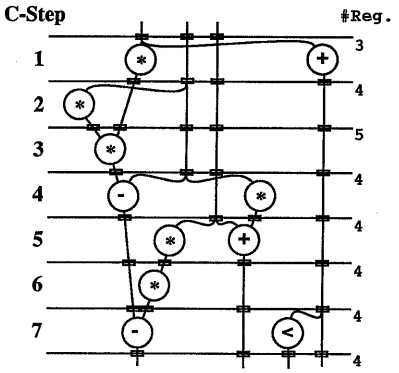


図 5: 繰り返し 1 回分のスケジューリング結果

このように、1 回の Look-ahead List スケジューリングでレジスタ数の制約を満たさなかった場合、次の段階として、変数をメモリに割り当てることにより、レジスタ数の減少を図る。まず、レジスタ数の制約を満たさなかった制御ステップに着目し、その制御ステップで保持する変数の中から、メモリに割り当てる変数を選ぶ。例題では、図 6の太い実線で示したエッジの中から選ぶことになる。これらのエッジのうち、演算の入出力として必要となる時間にもっとも余裕のある変数を選択する。図 6において、太い実線で示したエッジの右にある数字は、そのエッジの表す変数が、問題となっているステップの前後で演算の入出力として使われる制御ステップの間隔を示している。この場合、一番右側のエッジが、もっとも時間間隔の大きいエッジで

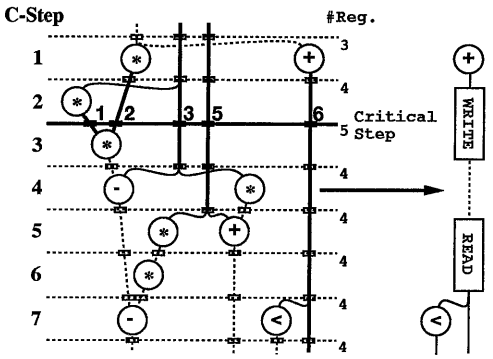


図 6: メモリに割り付ける変数の決定

ある。そこで、このエッジに、図6の右に示したようにメモリへの書き込みおよび読み出しのノードを一組挿入し、再度スケジューリングを行なう。その際、書き込みおよび読み出しノードには、図7に示したように、他のノードとの順序関係に制約を加える。すなわち、メモリ読み出しノードは、演算1,2の制御ステップ以降の制御ステップに割り当て、メモリ書き込みノードは、演算3,4,5,8,9より前の制御ステップに割り当てるという順序関係を与える。

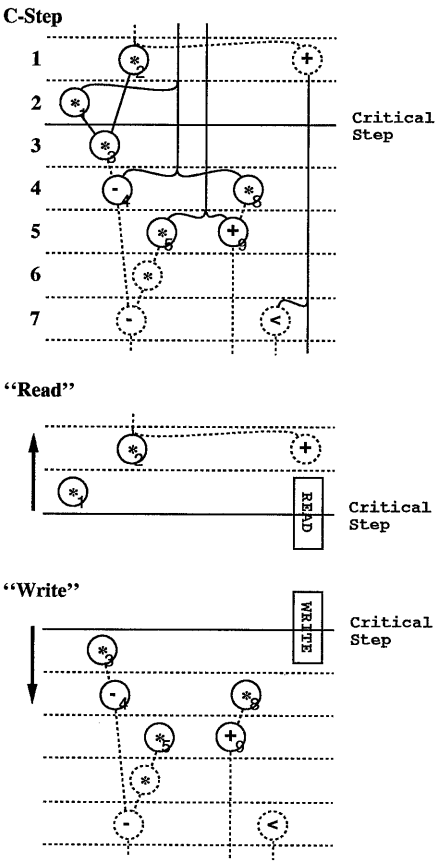


図 7: メモリのノードに対する順序関係

再度、Look-ahead List スケジューリングを行なった結果を図8に示す。この例題では、2回目でレジスタ数制約が満たされ、スケジューリングを終了する。

以上、例題を用いて述べてきたスケジューリングアルゴリズムを次に示す。はじめに、Look-ahead List スケジューリングのアルゴリズムを示す。

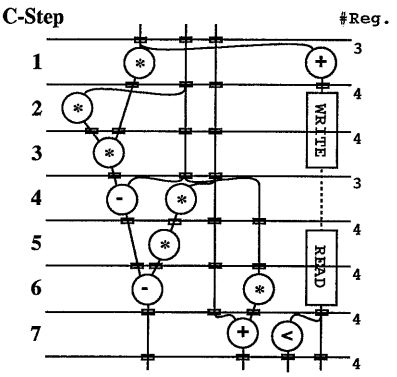


図 8: スケジューリング結果

Look-ahead List スケジューリングアルゴリズム

Step1 制御ステップ番号 i を $i=1$ に初期化する。

Step2 データ依存関係、演算の順序関係およびメモリ読み書きのタイミング制約から、制御ステップ i に割り当てることのできる演算を並べる。

Step3 演算器の種類と種類毎の個数の制約から、並べた演算の可能な組合せを挙げる。ただし、「何も割り当てない」という組合せは除く。挙げた組合せ各々について、次の2つを行なう。

- 必要となるレジスタ数を調べる。
- 割り当てる演算の“高さ”の大きい順に演算を並べる。

Step4 次の選択基準に従い、割り当ての組合せの中から1つを選ぶ。

1. レジスタ数の制約を満たすもの。ただし、すべての組合せにおいて、レジスタ数の制約を満たさない場合は、もっともレジスタ数の少ないもの。
2. 上の条件で選ばれた組合せの中から、演算の“高さ”を上から順に比較し、もっとも大きいもの。

Step5 すべての演算の割り当てが終わったか。まだならば、$i=i+1$ とし、Step2へ。

Step6 終了。

提案するスケジューリングアルゴリズム全体、すなわち、RMA スケジューリング法は、この Look-ahead List スケジューリング法を用いて、次のように表される。

RMA スケジューリングアルゴリズム

Step1 Look-ahead List スケジューリングを行なう。

Step2 すべての制御ステップにおいてレジスタ数の制約が満たされたか。満たされたならば、Step6 へ (終了)。

Step3 レジスタを最も多く必要とする制御ステップを挙げる。その制御ステップについて、その制御ステップと次の制御ステップとの区切り (ステップライン) をまたぐ変数エッジのうち、入出力される間隔がもっとも長いものを選び、そのエッジにメモリの書き込みノードおよび読み出しノードを挿入する。エッジが重複する場合は、メモリの読み書きノードは一組にまとめる。

Step4 問題となっているステップラインの上下の演算について、部分的に順序関係をつける。すなわち、

1. 書き込みノードと、当該ステップラインをまたぐ変数を入力とする演算ノード
2. 当該ステップラインをまたぐ変数を出力する演算ノードと読み出しノード

について、各々、前者のノードを先の制御ステップに割り当てる順序関係をつける。

Step5 Step1 へ。

Step6 終了。

3.2 計算量

提案手法すなわち RMA スケジューリング法およびその基本スケジューリングである Look-ahead List スケジューリング法は、各制御ステップにおいて割り当てが可能な演算の組合せをすべて調べるため、計算量が爆発する可能性を持っている。もし、すべての演算にデータを介した依存関係がまったくなく、演算器数, レジスタ数制約がともにすべての演算を同一制御ステップに割り当てることを許すものと仮定すると、組合せの数は $o(2^n)$ (n: 演算ノード数) になる。しかし、実際の動作記述では、演算の依存関係から並列に実行できる演算数は限られており、また、リソースの制約により、組合せの数はさらに制約を受けるため、計算量はさほど大きくならない。

3.3 性能の下限

RMA スケジューリング法はレジスタ数の制約を考慮に入れない場合、Look-ahead List スケジューリングを 1 回行なって終了するが、レジスタ数制約のないとき、Look-ahead List スケジューリング法 (LLS 法とする) は List スケジューリング法と同一の結果を得る。すなわち、次の命題が成り立つ。

命題 1 各制御ステップ S_i において、*List* スケジューリングによって割り当てられる演算の集合を A_i,*LLS* 法によって割り当てられる演算の集合を B_i としたとき、提案手法において演算器の種類と個数以外の制約を持たないならば、すべての制御ステップ S_i において

$$A_i = B_i$$

である。

命題の証明は付録に示す。この命題により、提案手法が演算器数, 制御ステップ数の面で、最悪でも List スケジューリング法と同等の性能をもつことが保証される。また、今回、優先基準として演算の "高さ" を用いたが、上の命題は、優先基準として何を用いるかに関わらず成立する。List スケジューリングで有効とされる優先基準を提案手法にそのまま適用することができる。

4 実験および評価

4.1 Look-ahead List スケジューリングの実験

ベンチマークデータとして Elliptical Wave Filter[8](以下、EWF とする) に対し、実験を行なった。まず、繰り返して行なうスケジューリングの 1 回分である Look-ahead List スケジューリングについて、実験結果を表 1, 表 2, 表 3に示す。

表 1: EWF に対するスケジューリング結果 1

	C-Step	× pl†	+	Reg.	CPU(sec)	比率‡
HAL[8]	18	2	3	12	120	1.00
OASIC[5]	17	2	3	10	30	1.00
JIT[9]	17	2	3	10	0.29	1.00
提案手法	17	2	3	10	0.4	1.00
HAL	18	1	3	12	240	2.00
OASIC	18	1	3	10	180	6.00
JIT	18	1	3	10	0.28	0.97
提案手法	18	1	3	10	0.4	1.00
HAL	19	1	2	12	360	3.00
OASIC	19	1	2	9	360	12.00
JIT	19	1	2	9	0.25	0.86
提案手法	19	1	2	10	0.4	1.00
提案手法	20	1	2	9	0.4	1.00

† 2 サイクルのパイプライン乗算器
‡ 比率 = 実行時間 / 最短ステップ数での実行時間
• CPU-time は手法により計算機環境が異なる。
HAL,OASIC — IBM PS/2 model 80
JIT — APOLLO 400 workstation
提案手法 — Sparcstation2

表 1は従来のスケジューリング法との比較である。ここで、OASIC[5] は制御ステップ数、演算器数, レジスタ数の点で最適な解を見い出している。提案手法はこれらの点で、ほぼ最適な解を得ている。実行時間で見ると、HAL[8], OASIC では、制御ステップ数が少し増えると、計算量が急増していく。これに対し、提案手法では制御ステップ数が増えても、計算量はあまり増加しない。

表 2は、最小のハードウェアリソースでスケジューリングを試みたものである。ここで、"FDS" は Force-Directed

スケジューリング法 [8] を我々が追試した結果である。参考として上段に、最短ステップ数での結果を挙げ、下段に最小リソース量での結果を挙げる。ハードウェアリソースを

表 2: EWF に対するスケジューリング結果 2

	C-Step	×	+	Reg.	CPU(sec)	比率
FDS	17	3	3	11	0.5	1.00
提案手法	17	3	3	10	0.4	1.00
FDS	28	1	2	11	30.5	61.00
FDS	33	1	1	11	47.0	94.00
提案手法	28	1	1	10	0.6	1.50

CPU-Time — FDS, 提案手法： Sparcstation2 を使用

少なくした場合、より多くの制御ステップ必要となっている。しかし、提案手法では非常に短い実行時間でスケジューリングを実現している。また、FDS 法と比較して、制御ステップ数, ハードウェアリソース量の点で、よりよい性能を得ている。

表 3は、“EWF” のループを数回展開した記述に対するスケジューリング結果である。提案手法は、演算数の増加

表 3: EWF に対するスケジューリング結果 3

ループ数	演算数	C-Step	× pl	+	Reg.	CPU	比率
2	68	34	1	3	10	1.2	2.40
3	102	50	1	3	10	3.0	6.00
4	136	66	1	3	10	6.0	12.00

• CPU-Time (単位：秒) — Sparcstation2 を使用

に従って計算量は増えてはいるが、急激な増加ではなく、100 を越える演算数でも、現実的な実行時間で実現している。制御ステップ数, 演算器数, レジスタ数はいずれの場合も最適である。

4.2 RMA スケジューリングの実験

RMA スケジューリングを “EWF” に対して行なった結果を表 4に示す。いずれの場合も、与えられた演算器数, レジスタ数の制約の下で、制御ステップ数の大きな増加もなくスケジューリングを実現している。また、実行時間も問題とならないレベルに留まっている。

表 4: EWF に対するスケジューリング結果 4

C-Step	× pl	+	Reg.	CPU(sec)
18	2	3	8	0.9
19	1	3	8	0.9
20	1	2	8	0.9
29	1	1	8	1.1
20	1	3	7	1.3
21	1	2	7	1.5
30	1	1	7	2.4

• CPU-Time — Sparcstation2 を使用

レジスタ数 7, 制御ステップ数 21 でスケジューリングを実現した結果を図 9に示す。細い点線で示したエッジがメ

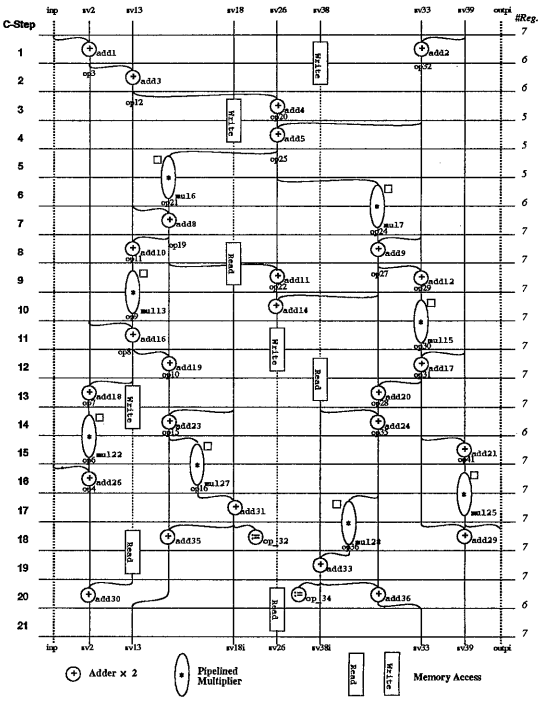


図 9: EWF に対するスケジューリング

モリに割り当てられた変数を表している。今回の実験では、データフローグラフへの入出力変数はあらかじめメモリに存在するわけではなく、最初のステップと最終ステップでは、レジスタに保持するものとして計算した。そのため、少なくとも入出力変数の数だけはレジスタを必要とする。例えば、“EWF” の例ではレジスタ数を 7 未満にすることはできない。しかし、入出力変数のデータは系の外部あるいはメモリ等に出し入れするものとすれば、この限りではなく、さらにレジスタ数を少なくすることができる。

5 まとめ

本稿では、動作記述中の変数の、メモリあるいはレジスタファイルへの割り付けを考慮したスケジューリング問題およびスケジューリング手法—RMA スケジューリング法を提案した。従来、スケジューリング時には、変数はすべてレジスタに割り当てることが前提とされていたが、提案手法により、メモリなどが混在した場合にもスケジューリングを行なうことができる。

RMA スケジューリング法は、演算器数などの制約に加えて、レジスタ数の制約および、メモリあるいはレジスタファイルのアクセス速度やポート数の制約を満たし、かつ、高速な処理を実現することが可能である。ベンチマークデータに対して実験を行ない、演算数あるいは制御ステップ数

が増大した場合でも、整数線形計画法などの手法と比較し、非常に短い実行時間で結果を得ることを確認した。また、RMA スケジューリング法は演算器数, 制御ステップ数の面で、最悪でも List スケジューリング法と同等の結果を得ることが保証されている。

課題として、レジスタおよびメモリ等のコスト評価法が挙げられる。実験例のように、1,2 個の変数のために新たにメモリなどを設けることは、ハードウェア量, 制御の複雑さを考えるとコストの最適化にはつながらない。最終的なコストの適切な評価が必要である。

謝辞

日頃から有益な助言を頂く NTT LSI 研究所 設計システム研究部 安達徹 主幹研究員ならびに 星野民夫 主幹研究員に深く感謝します。

参考文献

[1] M.C.McFarland,A.C.Parker and R.Camposano, *"Tutorial on High-Level Synthesis,"* Proc. of the 25th Design Automation Conference, pp.330-336,June 1988.

[2] M. Balakrishnan, A.K.Majumdar, D.K.Banerji, J.G. Linders and J.C.Majithia, *"Allocation of Multiport Memories in Data Path Synthesis,"* IEEE Trans. on Computer-Aided Design, Vol.7 No.4,pp.536-540,1988.

[3] C.-I.H.Chen and G.E.Sobelman, *"Singleport/Multiport Memory Synthesis in Data Path Design,"* Proc. of IEEE International Symposium on Circuits and Systems, pp.1110-1113, 1990.

[4] I.Ahmad and C.Y.Roger Chen, *"Post-Processor for Data Path Synthesis Using Multiport Memories,"* Proc. of IEEE International Conference on Computer-Aided Design (ICCAD-91), pp.276-279,1991.

[5] C.H.Gebotys and M.I.Elmasry, *"Optimal VLSI Architectural Synthesis: Area, Performance and Testability,"* Kluwer Academic Publishers,1992.

[6] M.Nourani and C.Papachristou, *"Move Frame Scheduling and Mixed Scheduling-Allocation for the Automated Synthesis of Digital Systems,"* Proc. of the 29th Design Automation Conference, pp.99-105,June 1992.

[7] F.Depuydt, G.Goosens and H.De Man, *"Clustering Techniques for Register Optimaization during Scheduling Preprocessing,"* Proc. of IEEE International Conference on Computer-Aided Design (ICCAD-91), pp.280-283,1991.

[8] P.G.Paulin and J.P.Knight, *"Force-Directed Scheduling for the Behavioral Synthesis of ASIC's,"* IEEE Transactions on CAD, Vol.8 No.6,pp.661-679,June 1989.

[9] Karl van Rompaey, Ivo Bolsens and Hugo De Man, *"Just In Time Scheduling,"* Proc. of IEEE International Conference on Computer Design (ICCD-92), pp.295-300, 1992.

付録

命題とその証明

命題 1 各制御ステップ S_i において、*List* スケジューリングによって割り当てられる演算の集合を A_i, *Look-ahead List* スケジューリング法 *(LLS 法)* によって割り当てられる演算の集合を B_i としたとき、*LLS* 法において演算器の種類と個数以外の制約を持たないならば、すべての制御ステップ S_i において

$$A_i = B_i$$

である。

証明. ひとつの制御ステップ S_i について証明すれば十分である。制御ステップ S_i に割り当てることが可能な演算の集合を T_i とする。演算 $o_j \in T_i$ のもつ優先順位を p_j とする。演算のもつ優先順位については、同じ優先順位をもつ演算はないと仮定する。すなわち、

$$j \neq k \implies p_j \neq p_k \tag{1}$$

である。(二つのスケジューリング法では、同じ優先順位をもつ演算が複数存在する場合があるが、その場合、先にリストに並んでいる演算を選択している。すなわち、先に並んでいる演算がより高い順位をもっているとすれば、上の仮定は成立する。) また、演算 o_j を割り当てることのできる演算器の数を n_j とする。

$(A_i \supset B_i)$　演算 o_j が $o_j \in B_i$ ならば、o_j と同じ種類の演算器に割り当てることができ、かつ、優先順位が p_j より高い演算 $(\in T_i)$ の個数は n_j より少ない。*List* スケジューリング法では優先順位が高い順に演算器数の制約が満たされなくなるまで演算を割り当てる。よって、演算 o_j は *List* スケジューリングによって制御ステップ S_i に割り当てられる。すなわち、$o_j \in A_i$ となり、$A_i \supset B_i$ が示される。

$(A_i \subset B_i)$　$o_j \in A_i$ であって、かつ $o_j \notin B_i$ なる演算 o_j が存在すると仮定する。このとき、式 *(1)* により、集合 B_i は、優先順位 p_j より高い優先順位をもつ演算の集合 $B1_i$ とより低い優先順位をもつ演算の集合 $B2_i$ に一意に分けられる。演算 o_j が $o_j \in A_i$ であることから、o_j より高い優先順位をもち、かつ、o_j と同じ種類の演算器に割り当てることができる演算 $(\in T_i)$ の個数は n_j より少ない。このことから、$B1_i$ に含まれる演算のうち、o_j と同じ演算器に割り当てることができる演算の個数は n_j より少ない。よって、演算の集合 $\{B1_i, o_j\}$ は提案手法において割り当てが可能な演算の組合せの一つである。演算の集合 $\{B1_i, B2_i\}$ すなわち B_i と $\{B1_i, o_j\}$ とを比較した場合、$\{B1_i, o_j\}$ の方がより高い優先順位をもち、提案手法によって集合 B_i が選ばれたことに矛盾する。よって、$o_j \in A_i$ かつ $o_j \notin B_i$ なる演算 o_j は存在しない。すなわち、$o_j \in A_i \Rightarrow o_j \in B_i$ であり、$A_i \subset B_i$ が示される。

以上のことから、$A_i = B_i$ が示された。∎