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

2006－SLDM－124（2）
2006／3／16

専用プロセッサ設計のためのレジスタ数を考慮した命令セット評価手法

増田　雅由　　伊藤　和人

埼玉大学大学院理工学研究科電気電子システム工学専攻
〒338-8570 埼玉県さいたま市桜区下大久保 255
E-mail: {masuda, kazuhito}@elc.ees.saitama-u.ac.jp

あらまし　プロセッサの命令セット選択は、プロセッサハードウェアとソフトウェア実行に対して、速度、面積、電力の面で大きな影響を及ぼす。アプリケーションに特化したプロセッサの設計において、命令セットを精度よく評価することが重要である。本論文では、与えられたアプリケーションに対して、レジスタ数制約によるメモリアクセスも考慮して高精度な命令セット評価を高速に行う手法を提案し、実験により有効性を確認する。

キーワード　専用プロセッサ、プロセッサ設計、命令セット評価、DAG カバーリング、分枝限定法

Instruction Set Evaluation with Register Constraint for Application Specific Processor Design

Masayuki MASUDA and Kazuhit ITO

Department of Electrical and Electronic Systems, Saitama University
255 Simoookubo, Sakura-ku, Saitama-shi, Saitama, 338-8570 Japan
E-mail: {masuda, kazuhito}@elc.ees.saitama-u.ac.jp

Abstract　The selection of instruction set of a processor greatly influences the processor hardware and execution of software in speed, area, and power. Evaluation of instruction set is an important task in designing a processor specific to a given application. In this paper, a technique to rapidly and precisely evaluate instruction sets for the given application is proposed with consideration of memory access by register constraint.

Key words　Application-specific processor, Processor design, Instruction-set evaluation, DAG covering, Branch and bound

1. 序　論

与えられたアプリケーション実行に最適化された専用プロセッサの設計では、ハードウェア構成（演算器の種類と数、レジスタ数、演算器間接続トポロジ）と命令セットを最適化する必要がある。ハードウェア構成ごとに最適な命令セットは異なる可能性があるので、個々のハードウェア構成候補に対して最適な命令セットを求めることで、最適なハードウェア構成と命令セットの組み合わせを得る必要がある。

そこでは、(1) 仮定したハードウェア構成について最適な命令セットを生成する手法、または (2) 仮定したハードウェア構成に適用可能な複数の命令セット候補から最適な命令セットを選択する手法が用いられている。(1) の従来手法として、アプリケーションのタスクスケジューリングを行い、その結果に基づいて命令セットを生成する手法が知られている。しかし、生成される命令セットが多数種類の命令を含み、命令実行が複雑なプロセッサとなる傾向がある。一方、(2) の手法では、従来は、各命令セット用にカスタマイズした汎用コンパイラを使用して実行ステップ数を求めることで命令セットを評価している。しかし、命令セット評価精度は、使用するコンパイラに強く依存する問題がある。

本研究では (2) の手法を採用し、仮定したハードウェア構成と命令セットについて、与えられたアプリケーションの最短実行ステップ数を DAG カバーリングによって求めることで命令セットを高速高精度に評価する手法を提案する。DAG カバーリングに基づく手法は [4] で提案されているが、レジスタ数が考慮されていないため、本研究では新たにレジスタ数を考慮した手法を提案する。

2. 問題定義

提案手法では、まずレジスタ数制約のないハードウェア構成に対して DAG カバーリングを用いて最短実行ステップ数を求める。DAG カバーリングは DSP の最適コード生成などに利用され、本研究においては高精度な命令セット評価を可能にする。

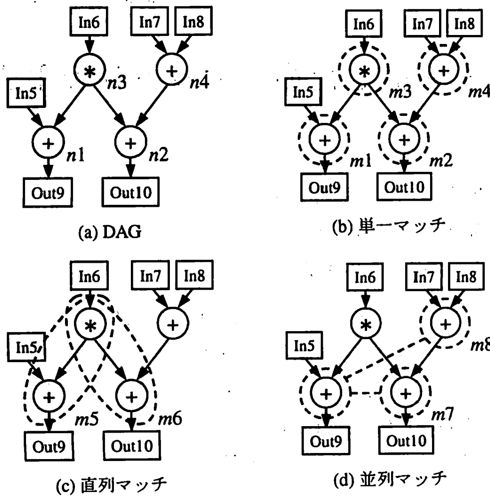


図 1　DAG と利用可能なマッチ

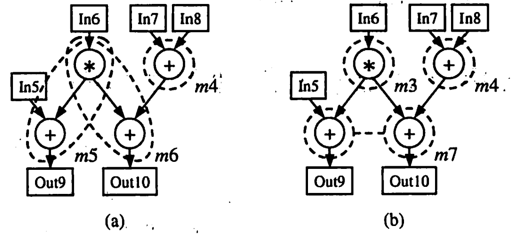


図 2　DAG カバーリング

(a)

CS	Match Schedule	Node Schedule M	A1	A2
5	m5	n3		
4				
3	m6	n3	n1	
2	m4		n4	
1			n2	

(b)

CS	Match Schedule	Node Schedule M	A1	A2
5				
4				
3	m3	n3		
2	m4		n4	
1	m7		n1	n2

図 3　図 2 の DAG カバーリングに対するスケジュール

2.1 DAG カバーリング問題

アプリケーション処理内容は有向無閉路グラフ（DAG）として与えられる（図 1 (a)）。DAG はノード（丸）、外部入出力（四角）、枝（矢印）からなり、それぞれ演算、外部との入出力、データ依存関係を表す。

命令は 1 つもしくは複数のノードを実行する。命令によって実行されるノードの集合をマッチと呼ぶ。DAG 中の各ノードは少なくとも 1 回実行（マッチによってカバー）する必要がある。DAG カバーリング [1] は DAG 全体をカバーするマッチの組み合わせとして定義される。DAG カバーリング問題は最小コスト DAG カバーリングを決定することである。

2.2 DAG カバーリングのコスト

命令セット評価問題において、DAG カバーリングのコストは DAG カバーリングに選択されたマッチを実行するために必要なステップ（クロックサイクル）数として定義する。

簡単な例を用いて DAG カバーリングとそのコストについて説明する。乗算器 1 個と加算器 2 個からなるハードウェア構成と、単一命令 ADD（加算）、単一命令 MUL（乗算）、直列命令 MUL・ADD（乗算後、加算を実行）、並列命令 ADD + ADD（2 つの加算を同時に実行）からなる命令セットを仮定する。この命令セットを用いるとき、図 1 (a) の DAG に対して、図 1 (b)-(d) の利用可能なマッチが導かれる。マッチ $m1, m2, m4$ は命令 ADD、$m3$ は MUL、$m5$ と $m6$ は MUL・ADD、$m7$ と $m8$ は ADD + ADD に対応する。

図 1 (a) の DAG に対して、図 1 (b)-(d) の利用可能なマッチを用いた 2 つの DAG カバーリングを図 2 に示す。各 DAG カバーリングについてリストスケジューリング [2] を施した結果を、それぞれ図 3 (a), (b) に示す。図 2 (a) の DAG カバーリングの実行に必要なステップ数は 5、(b) の実行ステップ数は 3 であり、より短い実行ステップ数を達成する図 2 (b) の方が良解である。

3. DAG カバーリング問題の解法

DAG カバーリング問題は後述する探索木に対する深さ優先探索によって解決される。

3.1 DAG カバーリング探索木

図 4 は、図 1 (a) の DAG と図 1 (b)-(d) の利用可能なマッチに対する正当な DAG カバーリングを発見するための探索木である。ここで、探索点とは探索木中の点を指す。

探索点の上部　探索点 V_k の上部にはマッチの要素 $m(V_k)$ が割り当てられる。$m(V_k)$ は、V_k の親に割り当てられたノード集合の先頭ノードを出力ノードとしてカバーする。

探索点の下部　探索点 V_k の下部には、ノード集合 $N(V_k)$ が割り当てられる。探索点 V_k の親を V'_k とすると、$N(V_k)$ は、$N(V'_k)$ からその先頭ノードを削除したものにマッチ $m(V_k)$ の入力ノードを追加したものである。

探索点 V_k は部分 DAG カバーリングに対応し、ルート探索点 V_r から V_k までの経路上に存在する各探索点に割り当てられたマッチが DAG カバーリングに選択されている。例えば、探索点 V_6 に対応する部分 DAG カバーリングは $M(V_6) = \{m1, m6\}$ である。

探索点 V_k においてマッチ $m(V_k)$ が並列マッチのとき、$m(V_k)$ と同じノードをカバーする単一マッチは不要となるため DAG カバーリングから削除する。図 4 において、探索点間の枝 (V_1, V_{10}) の横に記される $-m1$ は単一マッチ $m1$ が削除されることを表し、部分 DAG カバーリング $M(V_{10})$ 中に $m1$ は存在しない。

探索木の葉（図 4 では $V_4, V_5, V_8, V_9, V_{12}, V_{16}, V_{18}$）が正当な DAG カバーリングに対応する。

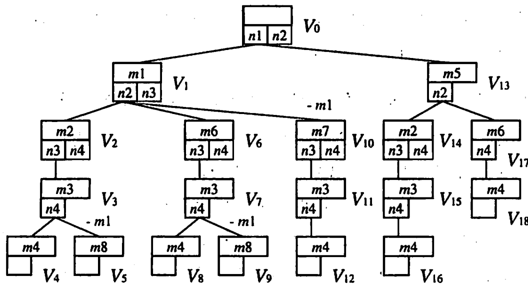


図 4　DAG カバーリング探索木

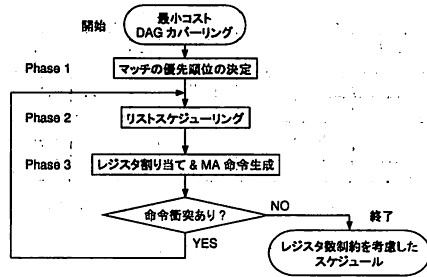


図 5　レジスタ数制約を考慮したスケジューリングアルゴリズム

3.2　分枝限定法

探索効率化のために分枝限定法を利用する。DAG カバーリング問題の性質を考慮して、次の基準によって枝刈りを行う。

DAG カバーリングの仮の最短実行ステップ数を TS（初期値は無限大）とする。TS は、探索が正当な DAG カバーリングを見つけ、その実行ステップ数が以前の TS 未満ならば更新される。探索点 V_k において、以下に示す $LB_M(V_k)$ と $LB_S(V_k)$ は、探索点 V_k を根とする部分探索木において発見される DAG カバーリングの実行ステップ数の下限である。いずれかの値が TS を超えるとき、探索点 V_k からの分枝を刈る。

独立集合選択によるマッチ数下限　1 ステップ 1 命令発行可能であるから、DAG カバーリングにおけるマッチ数は実行ステップ数の下限となる。独立集合 [3] の概念を利用して、探索点 V_k における部分 DAG カバーリング $M(V_k)$ およびノード集合 $N(V_k)$ から、将来（V_k を根とする部分探索木に対する探索中）生成される DAG カバーリングのマッチ数の下限を求め、$LB_M(V_k)$ とする。

スケジューリングによる実行ステップ数下限　探索点 V_k において、将来少なくとも 1 回はマッチによってカバーされる必要のあるノードの集合を $N_E(V_k)$ とする。$M(V_k)$ と $N_E(V_k)$ の実行に必要な実行ステップ数は、将来生成される DAG カバーリングの実行ステップ数の下限である。$M(V_k)$ と $N_E(V_k)$ を共にスケジュールすることによりそれらの実行に要する実行ステップ数を求め、$LB_S(V_k)$ とする。

3.3　下限に基づく優先探索

探索点 V_k における下限は、V_k を経由して発見される DAG カバーリングのコストの目安となる。下限のより小さい探索点から優先して探索することによって、早期に最小コスト DAG カバーリングを発見し、効率よく枝刈りを行う手法を提案する。

この手法では、DAG カバーリング探索木に対して幅優先探索を行う。なお、本研究では 2 種類の下限を使用しているため、探索点 V_k における両下限に基づいた優先関数を設定する。この優先関数は、直列マッチと並列マッチの性質を考慮して $S \times LB_M(V_k) + P \times LB_S V_k$ とする。ここで、S と P は命令セットに含まれる直列命令数と並列命令数である。探索点 V_k' を共通の親とする各探索点 V_k について、優先関数の値の小さい探索点から順に探索の優先順位を決定する。

4.　レジスタ数制約の考慮

同時に保持する必要のあるデータ数がレジスタ数を超える場合、データをメモリへ退避し（SPILL）、必要なときにレジスタへ復元する（RELOAD）。これら SPILL と RELOAD を合わせてメモリアクセス処理（MA 処理）と呼ぶことにする。MA 処理を実行するために、演算命令に加えてメモリアクセス命令（MA 命令）の発行が必要となる。したがって、高精度な命令セット評価のためには、レジスタ数制約を考慮した最短実行ステップ数を求める必要がある。

本研究では、レジスタ数制約のない最小コスト DAG カバーリングに対してレジスタ数制約を考慮したスケジューリングを施して実行ステップ数を求める手法を提案する。レジスタ数制約を考慮したスケジューリングに関して、新たに以下の制約を満たさなければならない。

制約 4: レジスタ数制約　レジスタ数を超えるデータは同時に保持することができない。

制約 5: MA 命令制約　MA 命令とマッチは同時に実行開始することができない。

なお、本研究で仮定するプロセッサは複数の MA 処理を同時に実行可能である。

図 5 に提案するスケジューリングアルゴリズムを示す。DAG カバーリングを与え、レジスタ数制約を考慮した実行可能なスケジュールを得る。各 Phase については以降の小節にて説明する。Phase 3 終了後に命令衝突検査を行い、制約 5 を満たしていればアルゴリズムを終了する。

提案手法では、DAG カバーリング中のマッチに対して MA 情報を付加することにより、命令衝突のない実行可能なスケジュールが得られる（アルゴリズムが収束する）ことを保証する。図 6 は、図 1 のマッチ $m6$ に関する情報を表す。図 6 (a) は、マッチ $m6$ の実行開始位置（行 Op. Inst.）を CS = 1 と仮定したとき、$m6$ の実行する各ノードの演算期間（行 ADD と MUL）と、演算に利用するデータおよび演算結果をレジスタに保持する必要のある期間（行 D2, D4, D6）を表す。乗算 $n3$ は CS = 1-2 で、加算 $n2$ は CS = 3 で実行される。D2, D4, D6 は、$n2$, $n4$, In6 の出力データを表す。D6 と D4 は、それぞれ $n3$ と $n2$ の実行に利用されるため、各演算期間中（各々 CS = 1-2 と

CS	0	1	2	3	4	5	6
Op. Inst.		m6					
MA Inst.							
ADD				n2			
MUL		n3					
D2							
D4							
D6							

(a) 基本情報

CS	0	1	2	3	4	5	6
Op. Inst.		m6					
MA Inst.	MA				MA		
ADD				n2			
MUL		n3					
D2					S		
D4							
D6	L						

(b) MA 情報

図 6　マッチ情報

CS = 3）レジスタに必ず保持しておかねばならない。また、D2 は $n2$ の実行結果として出力されるため、その演算直後（CS = 4）はレジスタに必ず格納しなければならない。このような期間をデータの拘束期間と呼ぶ。また、マッチは自身の持つデータ情報をその拘束期間中に要求するといい、例えば $m6$ は D6 を CS = 1-2 に、D4 を CS = 3 に、D2 を CS = 4 に要求するという。図 6 (a) はマッチが潜在的に持つ基本情報であり、各マッチが個々の基本情報を持つ。図 6 (b) は、D6 を CS = 0 で RELOAD（図中の L）、D2 を CS = 4 で SPILL（図中の S）する必要のあることを表す。このとき、CS = 0, 4 で MA 処理を実行するために MA 命令が生成され、行 MA Inst. の CS = 0, 4 に MA と示される。これを $m6$ に付加した MA 情報と呼ぶ。すなわち $m6$ は、その実行開始位置を基準として 1 ステップ前と 3 ステップ後の CS に MA 情報を持つ。

マッチの実行開始位置と自身の持つ MA 情報が同一 CS に存在するとき、命令衝突が生じることになる。このような場合、RELOAD は 1 ステップ前、SPILL は 1 ステップ後へ移動させることによって衝突を回避する。

4.1 Phase 1: 優先順位の決定

Phase 1 では、Phase 2（リストスケジューリング）のために DAG カバーリング中のマッチについて優先順位を決定する。Phase 2 を繰り返し実行する際に各マッチの位置関係ができるだけ変化しないよう、以下の基準に従って決定する。

1. マッチ間のデータ依存関係があるマッチ同士においては、その先行制約に従う。
2. マッチ間のデータ依存関係がないマッチ同士においては、レジスタ数制約を考慮しないリストスケジューリング結果に基づき、スケジュール CS の遅いマッチを優先する。

4.2 Phase 2: リストスケジューリング

Phase 2 では、Phase 1 で決定したマッチの優先順位に基づき、DAG カバーリングに対してリストスケジューリングを施す。ただし、ここではスケジューリング制約 1-3 [4] に加え、制約 4, 5 に関する以下の制約を考慮する。

制約 4'　各 CS において、拘束期間データ数はレジスタ数以下でなければならない。

制約 5'　MA 情報とマッチの実行開始位置が同一 CS に存在してはならない。

制約 4' を満足することは、制約 4 を満たすための必要条件である。一方、制約 5' は、MA 情報を利用してアルゴリズムを収束させるための手段である。

4.3 Phase 3: レジスタ割り当ておよび MA 命令生成

Phase 3 では、Phase 2 のスケジューリング結果に基づき、レジスタ割り当ておよび MA 命令生成を行う。レジスタ割り当てとは、各データ区間（拘束期間と自由期間）を各レジスタへ割り当てることをいう。ここで、自由期間とはデータの各拘束期間の間に発生するデータ保持期間である。レジスタに割り当てられた期間の間、そのデータはレジスタに格納されることを表す。したがって、データの拘束期間は必ずレジスタへ割り当てる必要がある。一方、自由期間は必ずしもレジスタへ割り当てる必要はないが、メモリへ格納する際に、そのデータに関して MA 処理が発生する可能性がある。MA 処理が生成されるたび、MA 命令生成および MA 情報付加がなされる。すべてのデータの拘束期間がレジスタへ割り当てられたとき、DAG カバーリングの実行に必要なすべての MA 命令が生成されている。

以下に、レジスタ割り当てアルゴリズムを示す。アルゴリズム中で使用する MA シフトとは、データ区間の割り当てによって生じる MA 処理をすでに MA 命令の割り当てられた CS、またはマッチの割り当てられていない CS へ移動することである。MA 処理が MA シフト可能な場合、新たな MA 命令は生成されないか、生成されたとしても演算命令と衝突せずに発行可能なため、この MA 情報はマッチへ付加しない。

1. レジスタに未割り当ての拘束期間のうち、開始 CS の最も若い拘束期間を含むデータ区間を T_{Data} とする。もし複数候補存在する場合は、メモリに格納されているデータ区間を優先し、なお決定しない場合はデータ番号最小のものを選択する。また、その開始 CS を *TCS* とする。
2. 以下の手順に従って T_{Data} をレジスタに割り当てる。
   (a) *TCS* にレジスタの空きがある場合
   T_{Data} を割り当てるレジスタ T_{Reg} を選択する（4.4 節）。
      i. MA 処理生成
      T_{Data} がメモリに格納されている場合は、*TCS* より前に T_{Data} を RELOAD する必要がある。このとき、*TCS* より前かつ T_{Reg} に割り当てられた拘束期間終了 CS 以降で、最も遅い MA シフト可能 CS にて RELOAD を行う。MA シフト不可能な場合は、*TCS*- 1 で RELOAD を行い、T_{Data} を *TCS* に要求するマッチに MA 情報（RELOAD）を付加する。
   T_{Data} にレジスタ割り当て済みの印を付ける。
   (b) *TCS* にレジスタの空きはないが、いずれかにデータの自由期間を含む場合
   T_{Data} を割り当てるレジスタ T_{Reg} を選択する（4.4 節）。
      i. データ区間分割およびレジスタ割り当て解除
      レジスタの空きを作るために、T_{Reg} のデータ区間のうち *TCS* を通過する自由期間を除去し、分割された前半データ区間を T'_{Pre} とする。残りの後半部分を T'_{Post} とし、レジスタ割り当てを解除する。
      ii. MA 処理生成
      T'_{Pre} がメモリに格納されていない場合は *TCS* より前に T'_{Pre} を SPILL し、T_{Data} がメモリに格納されている場合は *TCS* より前に T_{Data} を RELOAD する必要がある。

CS	0	1	2	3	4	5	6	7	8
Op. Inst.		A	M			A	M	A	M
MA Inst.	MA				MA				
R1		D10	D1					D1	
R2		D11			D6	D9			
D1									
D12 M									

(a)

CS	0	1	2	3	4	5	6	7	8
Op. Inst.		A	M			A	M	A	M
MA Inst.	MA				MA				
R1		D10	D1						
R2		D11			D6	D9			
D1									
D12 M									

(b)

CS	0	1	2	3	4	5	6	7	8
Op. Inst.		A	M			A	M	A	M
MA Inst.	MA				MA				
R1		D10	D1						
R2		D11			D6	D9			
D1 M					S				
D12 M									

(c)

CS	0	1	2	3	4	5	6	7	8
Op. Inst.		A	M			A	M	A	M
MA Inst.	MA				MA				
R1		D10	D1			D12			
R2		D11			D6	D9			
D1 M					S				
D12 M					L				

(d)

図 7　レジスタ割り当てアルゴリズム 2.(b) の実行手順

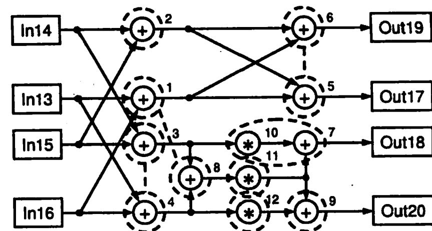
このとき、TCS より前かつ T'_{Pre} の拘束期間終了 CS 以降で、最も遅い MA シフト可能 CS にて MA 処理を行う。MA シフト不可能な場合は、まず T_{Data} の RELOAD が必要ならば TCS- 1 で行い、T_{Data} を TCS に要求するマッチへ MA 情報を付加する。T'_{Pre} の SPILL が必要ならば、先の結果を受けて、MA シフト可能であれば TCS- 1 で行う。MA シフト不可能であれば T'_{Pre} の拘束期間終了 CS で SPILL を行い、T'_{Pre} を T'_{Pre} の拘束期間終了 CS に要求するマッチへ MA 情報を付加する。T'_{Post} にメモリ格納済みの印を付ける。

T'_{Pre} にレジスタ割り当て済みの印を付ける。

(c)　TCS に空きがなく、すべてが拘束期間の場合

Phase 2 において制約 4' を課しているため、このような場合は存在しない。

図 7 に、簡単な例を用いて前述のアルゴリズム 2.(b) の実行手順を示す。図 7 (a) は開始時のレジスタ割り当て状態である。行 R1, R2 が各レジスタに割り当てられた各データのデータ区間であり、斜線の領域がその両端の拘束期間によって生じる自由期間を表す。また、行 D12 に記された M は、D12 のデータ区間がメモリに格納されていることを示す。ここで、TCS= 5, T_{Data}= D12, T_{Reg}= R1 であり、D12 を R1 へ割り当てる。まず、R1 のデータ区間 (D1) のうち、CS = 5 を通過する自由期間を除去し、分割された D1 の後半部分を未割り当てとする (図 7 (b))。D1 はメモリに格納されていないため、CS = 5 より前に SPILL する必要がある。CS = 5 より前かつ D1 の拘束期間終了 CS (CS = 3) 以降で、最も遅い MA シフト可能 CS (CS = 4) にて SPILL を行う (図 7 (c))。同時に、MA シフトによって D1 の拘束期間が CS = 3 までから CS = 4 へ延長されている。D1 の未割り当てデータ区間はメモリに格納されたため、行 D1 に M の印を付ける。最後に、D12 を R1 へ割り当てる (図 7 (d))。このとき、D12 はメモリに格納されているため RELOAD が必要であり、CS = 4 にて RELOAD を実行する。



$m2 = \{n11\}, m3 = \{n12\}, m5 = \{n2\}, m12 = \{n9\},$
$m13 = \{n10, n7\}, m21 = \{n1, n8\}, m28 = \{n3, n4\}, m33 = \{n5, n6\}$

図 8　4 次離散コサイン変換に対する DAG カバーリング

CS	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Op. Inst.		m28	m3	m21	m2		m5	m13			m33		m12	
MA Inst.	MA					MA				MA		MA		MA
M			n12		n11			n10						
A1		n3		n1			n2			n7	n5		n9	
A2		n4		n8							n6			
R1		D13		D13	D1		D11			D11			D11	D9
R2		D14	D3					D3			D1	D5	D12	
R3		D15	D4		D8		D14	D2			D2	D6		
R4		D16		D16	D12		D15				D7			

図 9　図 8 に対するスケジューリング結果

表 1　命令セット

IS	命令セットに含まれる演算命令
1	MUL, ADD
2	MUL, ADD, MUL・ADD,　ADD・ADD
3	MUL, ADD, MUL・ADD,　ADD + ADD
4	MUL, ADD, MUL + ADD, ADD + ADD

4.4 T_{Reg} の選択

T_{Reg} には、T_{Data} をレジスタに割り当てる際、新たに命令衝突の発生する可能性がより低いものを優先して選択する。まず、各レジスタ候補選択時の命令衝突発生回数を推定する。

1. レジスタ候補を C_{Reg}、命令衝突の発生回数を *Conflicts* (初期値は 0) とする。
3. 以下の手順によって *Conflicts* を計算する。
   (a) T_{Data} がメモリに格納されていない場合、
      i. TCS において、C_{Reg} に自由期間が割り当てられているならば、*Conflicts* += 0.5 とする。
      ii. i. の条件、かつそのデータ区間がメモリに格納されていない、かつ MA シフト不可能ならば、*Conflicts* += 1 とする。
   (b) T_{Data} がメモリに格納されている場合、
      i. TCS において、C_{Reg} に自由期間が割り当てられているならば、*Conflicts* += 0.5 とする。
      ii. MA シフト不可能ならば、*Conflicts* += 1 とする。

表 2 DAG カバーリング結果

		従来手法				提案手法					
		1. 分枝限定法		2. 高速化手法 [4]		3. 分枝限定法			4. 高速化手法		
DAG	IS	#S	CPU [sec]	#S	CPU [sec]	#S	CPU [sec]	探索点数	#S	CPU [sec]	探索点数
WEF	1	35	0	35	0	35	0	34	35	0	34
	2	24	54	25	28	24	8	295,294	24	1	42,209
	3	NA	NA	22	14	22	0	20,393	22	0	332
	4	NA	NA	22	342	21	0	2,736	21	0	534
DCT	1	40	0	40	0	40	0	40	40	0	40
	2	32	22,829	33	6,159	31	40,160	1,129,190,655	31	505	18,489,739
	3	NA	NA	25	32,892	24	933	87,617,664	24	368	34,286,002
	4	NA	NA	23	40,310	23	1	106,048	23	0	1,104

各レジスタ候補のうち、最小衝突回数のレジスタを *TReg* に選択する。衝突回数最小となるレジスタが複数存在するときは、以下の基準に従う。

1. 最小 *Conflicts* = 0 または 1 の場合、*TCS* において各レジスタ候補は必ず空である。このとき、レジスタ番号最小のものを選択する。
2. 最小 *Conflicts* = 0.5 または 1.5 の場合、*TCS* において各レジスタ候補には必ず自由期間が割り当てられている。このとき、その自由期間の終了 CS が最も遅いレジスタを優先する。なお決定しない場合は、レジスタ番号最小のものを選択する。

図 8 は、4 次離散コサイン変換に対する DAG カバーリング例である。レジスタ数 4 個を仮定したとき、これに対するスケジューリングアルゴリズム（図 5）の実行結果を図 9 に示す。

5. 実験結果

提案する命令セット評価手法を C++ 言語を用いて実装し、2GHz PC 上で実行した。DAG として 5 次ウェーブディジタル楕円フィルタ（WEF）と 8 次離散コサイン変換（DCT）を与え、乗算器 1 個と加算器 2 個からなるハードウェア構成を仮定する。これらに対して 4 種類の命令セット（表 1）を仮定し、各々について命令セット評価を行う。

表 2 に DAG カバーリング結果を示す。#S はレジスタ数制約のない最短実行ステップ数を表す。従来手法 1 と提案手法 3 は DAG カバーリング探索木の構成が異なり、従来手法 2 は従来手法 1 に並列マッチ削減と DAG 分割による高速化手法 [4] を組み込んだものである。この結果は、提案手法 3 によって、従来手法 1, 2 よりも高精度な解が実用的な時間内で得られている。更に、提案手法 3 に独立集合選択によるマッチ数下限計算と下限に基づく優先探索を組み込んだ提案手法 4 では、解の最適性を損なうことなく CPU 時間を大きく削減することができる。

表 3 にレジスタ数制約を考慮したスケジューリング結果を示す。いずれの結果も、CPU 時間 1 秒未満で得られた。#M はマッチ（演算命令）数、#R はレジスタ数を示す。#S' と #MA は、#R を仮定したときの実行ステップ数と MA 命令数である。各命令セットについて、最上行のレジスタ数がその命令セットを用いるときに最低限必要なレジスタ数であり、* の付いたレジスタ数は最短実行ステップ数を達成する最小レジスタ数を表す。この結果は、MA シフトを用いることによって MA 命令数と実行ステップ数が大幅に削減されることを示す。

表 3 レジスタ数制約を考慮した命令セット評価

				MA シフトなし			MA シフトあり		
DAG	IS	#S	#M	#R	#S'	#MA	#R	#S'	#MA
WEF	1	35	34	2	70	31	2	70	31
				*9	47	12	9	40	5
				11	47	12	*11	36	2
	2	24	23	3	58	27	3	50	20
				*9	35	9	9	28	3
				11	35	9	*11	25	2
	3	22	21	4	46	19	4	38	14
				*10	34	9	10	26	4
				12	34	9	*12	23	2
	4	21	20	4	43	18	4	38	15
				*9	31	10	9	26	4
				11	31	10	*11	22	2

結果的に、仮定した命令セットのうち、実行ステップ数最短となる命令セット 4 が最適命令セットである。

6. 結　　論

本論文では、与えられたアプリケーション処理と仮定したハードウェア構成に対する高速かつ高精度な命令セット評価手法を提案し、実験によりその有効性を確認した。提案手法により短時間の命令セット評価が可能となるため、複数の命令セット候補に対して評価を行うことによってハードウェア構成に最適な命令セットを選択する手法が実用化できる。

今後、適切な命令セット候補およびハードウェア構成候補を自動的に生成する手法を検討し、アプリケーションに特化した専用プロセッサの設計自動化を目指す。

文　　献

[1] S. Liao, et al., "Instruction Selection Using Binate Covering for Code Size Optimization", *Proc. Int. Conf. on Computer-Aided Design*, pp. 393-399, 1995.

[2] M. C. McFarland, et al., "The High-Level Synthesis of Digital Systems", *Proc. IEEE*, vol. 78, no. 2, pp. 301-318, 1990.

[3] O. Coudert, et al., "New Ideas for Solving Covering Problems", *Proc. 32nd DAC*, pp. 641-646, 1995.

[4] M. Masuda and K. Ito, "Rapid and Precise Instruction Set Evaluation for Application Specific Processor Design", *Proc. ISCAS 2005*, pp. 6210-6213, 2005.