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

2004－SLDM－117（3）
2004／12／1

専用プロセッサの命令セット評価の高速化手法

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，精度の高い命令セット評価を行うことが重要である．本論文では，与えられたアプリケーションに対して高精度な命令セット評価を高速に行う手法を提案する．実験により，提案する手法が効率的に命令セットを評価することを示す．

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

Rapid Instruction Set Evaluation for Application Specific Processor

Masayuki MASUDA　　Kazuhito ITO

Department of Electrical and Electronic Systems, Saitama University
255 Simookubo, Sakura-ku, Saitama-shi, Saitama, 338-8670, 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. The results show the proposed technique efficiently evaluate instruction sets for assumed processor hardware.

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

1. 序論

プロセッサの設計および合成のための EDA ツール，カスタムプロセッサの生成手法[1-3]，およびプロセッサのソフトウェア開発システムの生成手法[1,4,5]の発展に伴い，特定のアプリケーションに特化したプロセッサの設計が実用的になりつつある．

一般に，プロセッサの構成には多くの選択肢がある．ここで，構成とは，演算器の種類と数，レジスタ数，これら要素の相互配線，パイプラインステージ数などの組み合わせである．命令セットもまたプロセッサの構成の重要な一部となる．特定のアプリケーションについて，各構成は実行速度，チップ面積，電力消費の点で異なる性能を示す．これら選択肢となる構成の中から，速度，面積，電力の要求に対して最も適当な構成を選択し，プロセッサが実装される．

与えられたアプリケーションに対する最適な構成を直接生成することは非常に困難なため，複数の候補を生成，評価して最適なものを選択する．この評価はハードウェア評価とソフトウェア評価からなる．ハードウェア評価は，プロセッサ回路と物理設計(配置配線)の合成を意味し，プロセッサのクロック速度とチップ面積を決定する．ソフトウェア評価は，与えられたアプリケーション内のタスクのスケジューリングを意味し，アプリケーションの実行速度とプログラムメモリおよびワークメモリのチップ面積を決定する．ソフトウェア評価には，プロセッサハードウェアとタスクのスケジュールに基づいて電力消費を評価するためのシミュレーションを含むこともある．

本研究では，ソフトウェア評価において，与えられたアプリケーションと仮定したハードウェア構成に対する命令セットを仮定する．そして，命令セット制約を考慮して，アプリケーション内のタスク(演算，条件分岐など)が最短実行ステップ数を達成するようにスケジュールする．命令セットの仮定は，タスクのスケジューリングに制約を与える．例えば，命令セットの定義によって同時に実行開始できる演算数が制限される．ゆえに，命令セットの選択はプロセッサの最終的な性能に大きな影響を及ぼす．このソフトウェア評価は命令セットの評価を意味している．高精度の命令セ

ットの評価では，与えられた命令セット制約下でのタスクスケジューリングにより，できるだけ精度良くアプリケーション実行ステップ数を決定することが極めて重要である．

本論文では，仮定したハードウェア構成と仮定した命令セットにおいて，与えられたアプリケーションの実行ステップ数を高速かつ高精度に決定するための手法を提案する．

2. 関連研究

ソフトウェア評価に注目すると，既存の研究は2つの種類に分類することができる．一方は，設計したプロセッサ用にカスタマイズされた汎用コンパイラを用いてコンパイルする手法である[1]．最適性は使用する汎用コンパイラに強く依存する．他方は，アプリケーション内のタスクのスケジューリングを行い，そのとおりにタスクを実行するための命令セットを生成する手法である[4],[5]．これは結果的に大きな命令セットを生じる傾向にある．

[6]におけるカバーリング問題は，DSPのような特殊なアーキテクチャをもつプロセッサのための最適コンパイルに用いられる．このアプローチは精度の高い命令セット評価に利用することができるが，膨大なCPU時間を要する．そのため，カバーリング問題の解法を高速化するための手法が必要不可欠である．

3. 問題定義

3.1. DAG

アプリケーション処理内容はデータフローグラフ(DFG)として与えられる．DFGはノードと枝からなり，それぞれ演算とデータ依存関係を表す．DFGはその処理が無限に繰り返し実行されることを暗に表現している．枝には繰り返し間データ依存を表す遅延をもつこともある．図1(a)は，4つの演算ノードn1, n2, n3, n4からなるDFGの例である．'In'は隣接する演算ノードへ外部からデータが入力されることを示し，'Out'は隣接する演算ノードから外部へデータが出力されることを示す．'D'は遅延を意味し，遅延をもつ枝の横に示される．図1(a)のDFGにおいて，ノードn1, n3, n4へ外部からデータが入力され，n2から外部へデータが出力される．また，枝(n1, n4)は遅延をもつ．

DFGは，高級プログラミング言語で記述されたソフトウェアのデータフロー解析，あるいはディジタル信号処理アルゴリズムから直接得られる．DFGから，遅延をもつ枝を除去し，新たな'In'と'Out'を生成することによって有向非巡回グラフ(DAG)へ変換する．図1(a)のDFGは図1(b)のDAGへ変換される．図1(a)において，遅延をもつ枝(n1, n4)が除去される．そして，ノードn4へデータが入力されることを示す'In'と，n1からデータが出力されることを示す'Out'，およびこれらを接続するための枝が生成される．図1(b)のDAGへ変換することにより，外部へデータを出力するノードはn1とn2へ変化する．

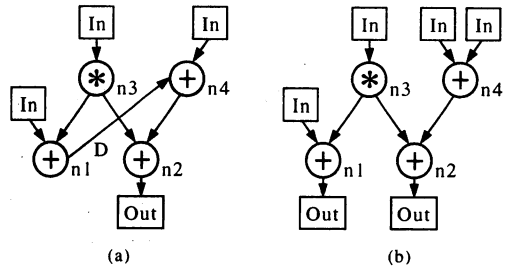

図1 (a) DFG (b) DFGから変換したDAG

3.2. DAGカバーリング

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，実行されなければならない．つまり，各ノードは少なくとも1つのマッチがカバーしなければならない．DAGカバーリング[6]は，すべてのノードを1回以上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適DAGカバーリングを決定することである．

3.3. スケジューリング制約

正当なDAGカバーリング内のマッチのスケジュールに関して，以下の制約を満足しなければならない．

制約1: 逐次命令実行
各実行ステップにおいて，たかだか1命令のみが実行開始可能である．これはプロセッサの命令実行における基本的な制約である．

制約2: データ依存関係による先行制約
各ノードnは，ノードnの実行に必要なデータを生成するノードが少なくとも1回，実行完了するまで実行開始してはならない．

制約3: ハードウェア制約
演算器数より多い数のノードは同時に実行することができない．

3.4. 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カバーリングに含まれる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．簡単な例を用いてこれを説明する．

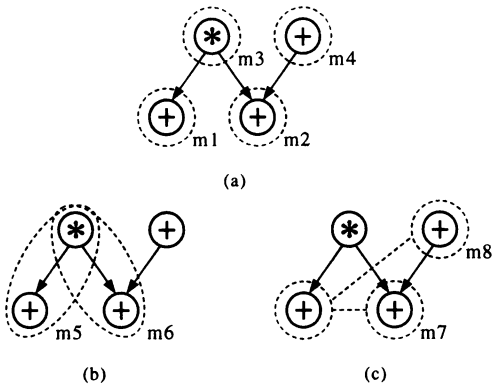

図 2　利用可能なマッチ (a) 単一マッチ
(b) 直列マッチ (c) 並列マッチ

プロセッサのハードウェア構成は乗算器 1 個と加算器 2 個からなるものと仮定する．この制約下で，命令 ADD(1 つの加算を実行)，MUL(1 つの乗算を実行)，MUL･ADD(乗算後，加算を実行)，および ADD+ADD(2 つの加算を並列に実行)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(b)の DAG に対する利用可能なマッチを図 2 に示す．マッチ m1, m2, m4 は命令 ADD に対応し，m3 は MUL，m5 と m6 は MUL･ADD，m7 と m8 は ADD+ADD に対応する．ここで，1 つのノードを実行するマッチを単一マッチ(図 2(a))，複数のノードを連続して実行するマッチを直列マッチ(図 2(b))，複数のノードを並列に実行するマッチを並列マッチ(図 2(c))と呼ぶ．マッチ m の入力ノードを，マッチ m 内のいずれかのノードへデータを供給するマッチ外のノードと定義する．各マッチの実行には，対応するすべての入力ノードの実行結果が必要である．マッチ m の出力ノードを，マッチ m 内で最後に実行されるノードと定義する．出力ノードからのみ，マッチ外へデータを出力できる．例えば，マッチ m6 の入力ノードは n4，出力ノードは n2 である．また，例えば，マッチ m1, m5, m7, m8 はそれぞれノード n1 を出力ノードとしてカバーすると言う．

図 1(b)の DAG について，図 2 の利用可能なマッチによる 2 つの DAG カバーリングを図 3 に示す．図 3(a), (b)において，すべてのノードが少なくとも 1 つのマッチによりカバーされているため，2 つの DAG カバーリングは正当である．結果として，図 3 の各 DAG カバーリングは 3 つの命令からなるプログラムを生じる．

図 3(a), (b)の DAG カバーリングについて，リストスケジューリング[7]を用いてマッチをスケジュールした結果を，それぞれ図 4(a), (b)に示す．図 4(a)のスケジュールでは，マッチは m6, m5, m4 の順に以下のようにスケジュールされる．

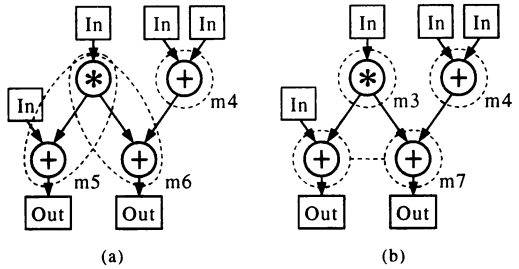

図 3　DAG カバーリング

Step	Match start	Node schedule M	A1	A2
-4	m5	n3		
-3				
-2	m6	n3	n1	
-1	m4		n4	
0			n2	

(a)

Step	Match start	Node schedule M	A1	A2
-4				
-3				
-2	m3	n3		
-1	m4		n4	
0	m7		n1	n2

(b)

図 4　DAG カバーリングのスケジュール

マッチm6: 乗算と加算の実行に 3 ステップを要するので step-2 で実行開始される．

マッチm5: スケジュール済みのノードとのデータ依存関係はなく，m6 と同様に step-2 で開始できる．しかし，制約 1(step-2 におけるマッチの実行開始数制限)，および制約 3(step-1 と step-2 における乗算器の利用数制限)のために m5 の実行は繰り上げられ，すべての制約を満足する step-4 で開始される．

マッチm4: step0 において加算器は 1 個しか利用されていないため step0 で開始できるが，制約 2(ノード n4 と n2 間のデータ依存制約)のために m4 は step -1 で開始される．

結果として，この DAG カバーリングの実行に必要なステップ数は 5 となる．一方，図 4(b)のスケジュールでは，マッチは m7, m4, m3 の順にスケジュールされる．結果として，この DAG カバーリングに必要なステップ数は 3 となる．したがって，図 3 の 2 つの DAG カバーリングにおいて，より短い実行ステップ数を達成する図 3(b)の方が良解である．

4. 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．ここで，探索点と

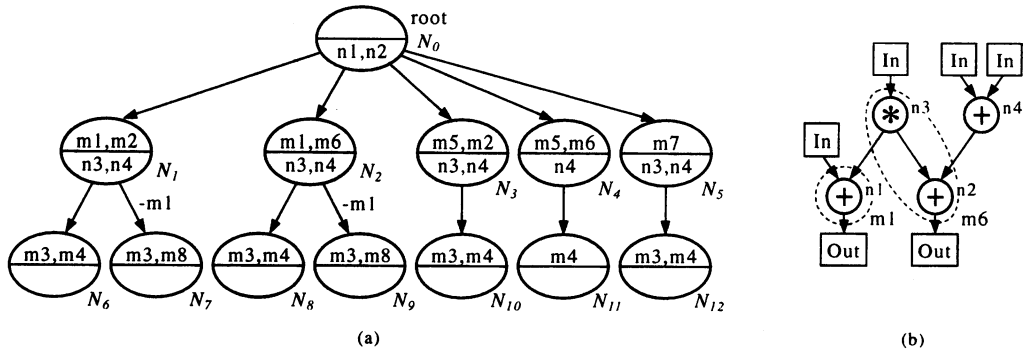

図 5　(a) DAG カバーリング探索木 (b) 探索点 N_2 に対応する部分 DAG カバーリング $M(N_2)$

は探索木中の点を意味し，図 5(a)に示すように探索点の下部は DAG 中のノードの集合，上部はマッチの集合を示す．

図 5(a)は，図 1(b)の DAG と図 2 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．以下に探索木の構成を説明する．

- root 探索点 N_0 は，DAG の外部へデータを出力するノードに対応する．N_0 において，DAG カバーリングのためのマッチは 1 つも選択されていない．
- N_0 を除いて，各探索点には 1 つもしくは複数のマッチが割り当てられる．これらのマッチは，先行する 1 つ前の探索点に割り当てられたすべてのノードを出力ノードとしてカバーする．
- マッチの割り当てられた探索点には，それらマッチの入力ノードが割り当てられる．ただし，その探索点に先行するすべての探索点のいずれかにおいて，すでに割り当てられているノードは考慮しない．

探索が探索点 N_x に到達したとき，探索点 N_x とそれに先行するすべての探索点に割り当てられたマッチが DAG カバーリングのために選択されているとする．例えば，探索点 N_6 ではマッチ m1, m2, m3, m4 が選択されている．言い換えれば，各探索点は部分 DAG カバーリング(または完全な DAG カバーリング)に対応する．ここで，探索点 N_x に対応する部分 DAG カバーリングに含まれるマッチの集合を $M(N_x)$とする．

図 5(b)は，図 5(a)の探索点 N_2 に対応する部分 DAG カバーリングであり，$M(N_2)$={m1, m6}である．N_2 に続く探索点 N_8 において，ノード n3 を実行する単一マッチ m3 が割り当てられることに注意する．ノード n3 は，探索点 N_2 においてマッチ m6 によってすでにカバーされている．マッチ m6 の実行において，ノード n3 は m6 の MUL 部分として実行され，その結果はノード n2 のみへ送られる．マッチ m1(ノード n1)の実行にもまた n3 の結果を必要とするが，m6 による n3 の実行結果は n1 には提供されない．ノード n1 用に n3 の結果を得るために，n3 はマッチ m3 によって再び実行する必要がある．これが探索点 N_8 にマッチ m3 が割り当てられる理由である．なお，他の方法としてノード n3 と n1 を 1 つの命令 MUL･ADD(マッチ m5)によって実行してもよい．これは経路(N_0, N_4, N_{11})に対応する．

探索点 N_x において，(1)並列マッチ内のすべてのノードが，N_x に先行するすべての探索点のいずれかに割り当てられ，(2)各ノードが未カバーもしくは単一マッチによりカバーされ，かつ(3)有向閉路(d-cycle)[6]を生じないときに限って，並列マッチが選択される．このときの単一マッチは，並列マッチの選択に伴って不要となるので DAG カバーリングから削除する．図 5(a)では，探索点 N_7 に並列マッチ m8 が割り当てられる．並列マッチ m8 内のノードは n1 と n4 であり，それぞれ探索点 N_0 と N_1 に割り当てられる．ここで，ノード n1 は単一マッチ m1 によりカバーされ，n4 は未カバーである．したがって，探索点 N_7 には並列マッチ m8 を割り当てることができる．並列マッチ m8 の選択に伴い，単一マッチ m1 は DAG カバーリングから削除され，$M(N_7)$={m2, m3, m8}となる．図 5(a)において，単一マッチ m1 の削除は探索点間の枝(N_1, N_7)の横に'-m1'として示されている．並列マッチ選択に関するこの制約は，各 DAG カバーリング間における探索点の共有を促すため，探索木の拡大防止に大きな効果をもつ．

探索が，下部が空の探索点(例えば，探索点 N_6)に到達した場合，すべてのノードがいずれかのマッチによってカバーされている．つまり，正当なカバーリングが発見されたことになる．図 5(a)で示されるように，図 1(b)の DAG と図 2 の利用可能なマッチに対して，7 つの正当な DAG カバーリング(N_6-N_{12})が存在する．

5. DAGカバーリングのための分枝限定法

実行可能なマッチの組み合わせが数多く存在するため，DAGカバーリング探索木は容易に増大する．探索効率化のため分枝限定法を利用する．分枝限定法は，見込みのない解候補へ導く枝を刈ることによって，探索木の不要な調査を省く手法である．DAGカバーリング問題の性質を考慮して，以下に示す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S(初期値は無限大)とする．Sは，探索が正当なDAGカバーリングに到達し，かつその実行ステップ数が以前のS未満ならば更新される．探索点N_xにおいて，以下に示す$|M(N_x)|$, P_x, L_xは，部分DAGカバーリング$M(N_x)$を含むすべての正当なDAGカバーリングの実行ステップ数の下限である．よって，いずれかの値がSを超えるとき，探索はN_xの先へ進まずにバックトラックする．これは探索点N_xから出るすべての枝を刈ることに対応する．

5.1. マッチ数 $|M(N_x)|$

マッチは命令に対応する．1ステップに1命令のみ実行開始可能であるから，マッチ数KのDAGカバーリングの実行にはK以上のステップ数を要する．探索点N_xではマッチ$M(N_x)$が選択されており，マッチ数は$|M(N_x)|$である．

5.2. 最長経路長 P_x

探索点N_xにおいて，ノードの演算時間を重みとしたDAGの最長経路長をP_xとする．並列マッチの選択により，ノードの並列実行を制約するための架空の枝がDAGに付加されるので，P_xは$M(N_x)$中の並列マッチに依存する．最長経路長P_xのDAGカバーリングの実行にはP_x以上のステップ数を要する．

5.3. 推定下限値 L_x

$Q_N(N_x)$を，探索点N_x以降の探索において，少なくとも1回は実行(マッチによりカバー)されなければならないノードの集合とする．図5(a)の探索点N_2(図5(b)の部分DAGカバーリング$M(N_2)$)において，$Q_N(N_2)$={n3, n4}である．ノードn3は$M(N_2)$中のマッチm6によってカバーされているが，n1用に再度実行する必要があるので$Q_N(N_2)$に含まれている．

部分DAGカバーリング$M(N_x)$を含む正当なDAGカバーリングの実行ステップ数を推定するために，マッチ$M(N_x)$とノード$Q_N(N_x)$を共にスケジュールする．ただし，$M(N_x)$中の各マッチについては3.3節で示されるすべての制約1, 2, 3を満足し，$Q_N(N_x)$中の各ノードについては制約2, 3のみを満足するようにスケジュールし，その実行ステップ数をL_xとする．ノード$Q_N(N_x)$をカバーする将来のマッチについては制約1が考慮されていない．加えて，$Q_N(N_x)$中のノードが将来2回以上実行されることを考慮しない．したがって，$M(N_x)$を含むすべての正当なDAGカバーリングの実行ステップ数の下限はL_xとなる．

特に，root探索点N_0における推定下限値を，初期推定下限値L_0とする．L_0は，生成され得るすべてのDAGカバーリングの実行ステップ数の下限である．実行ステップ数L_0を達成するDAGカバーリングが存在するならば，それは最適DAGカバーリングである．ゆえに，SがL_0を達成した時点で探索は終了する．

6. DAGカバーリングの高速解法

上述の分枝限定法は解の最適性を保証するものであるが，問題のサイズによっては実用時間内に解を得ることができない．探索の更なる高速化のために，解の最適性を損なう可能性をもちながらも，以下の手法を提案する．

6.1. 並列マッチの削減

並列マッチは，d-cycleを生じない限り，すべてDAGカバーリングに利用可能である．例えば，命令ADD+ADDが命令セット中に存在すれば，加算ノードの独立したすべてのペアが並列マッチとして考慮される．しかしながら，不用意な並列マッチの選択は長い経路長を導き，結果的にDAGの長い実行ステップ数を生じる可能性がある．そのような長い実行ステップ数へ導く並列マッチは選択すべきではなく，探索の進行において除外されるはずである(5.2節参照)．

DAGの'In'ノードからノードnへの最長経路長を$l(n)$とする．並列マッチ内のノードのすべてのペア(n_1, n_2)について，$|l(n_1)-l(n_2)| \leqq C$(Cは定数パラメータ)を満足する並列マッチのみ，DAGカバーリングにおいて考慮し，これ以外の並列マッチは除外する．

6.2. DAGの分割

DAG上に生成されるマッチ数の増加に伴い，DAGカバーリングに要する計算時間は指数関数的に増大する．しかしながら，膨大なマッチの少数の部分集合であれば，DAGカバーリングにおいて扱うことができる．与えられたDAGを分割して各部分DAGについてDAGカバーリングを施し，その結果を統合して全体DAGカバーリングとする手法を提案する．DAGカバーリングにおいて，対象部分DAG中のマッチのみを考慮するため，本来可能だが除外される直列マッチと並列マッチが発生する．そのため，解の最適性を損なう可能性があるが，求解時間を大きく削減可能である．

表1 命令セット

IS	命令セットが含む命令
1	MUL, ADD
2	MUL, ADD, MUL･ADD, ADD･ADD
3	MUL, ADD, MUL･ADD, ADD+ADD
4	MUL, ADD, MUL+ADD, ADD+ADD

表2 DAGカバーリング結果

		基本 分枝限定法			並列マッチ削減		
DAG	IS	#M	#S	CPU	#M	#S	CPU
4DCT	1	12	12	0sec	12	12	0sec
	2	22	10	0sec	22	10	0sec
	3	40	8	0sec	32	8	0sec
	4	54	8	0sec	44	8	0sec
WEF	1	34	35	0sec	34	35	0sec
	2	73	25	78sec	73	25	78sec
	3	190	NA	NA	98	22	12sec
	4	284	NA	NA	132	22	27sec

表3 DAG分割を利用したDAGカバーリング結果

		DAG 分割なし			DAG 分割あり	
DAG	IS	#M	#S	CPU	#S	CPU
WEF	1	34	35	0sec	35	0sec
	2	73	25	78sec	25	27sec
	3	98	22	12sec	22	13sec
	4	132	22	27sec	22	342sec
8DCT	1	40	40	0sec	40	0sec
	2	82	33	739min	33	42min
	3	216	NA	NA	25	548min
	4	368	NA	NA	23	671min

7. 実験結果

提案するソフトウェア評価手法をC言語を用いて実装し，2GHzプロセッサのPC上で実行した．対象DAGは5次ウェーブ楕円フィルタ(WEF)[8]，および8次離散コサイン変換(8DCT)[9]とする．WEFは加算26個と乗算8個からなり，8DCTは加算29個と乗算11個からなる．さらに8DCTから，4DCT(加算9個と乗算3個)を抽出する．プロセッサのハードウェア構成は，3.4節の例と同様である．これらのDAGとハードウェア構成に対して，表1に示す4種類の命令セットを仮定し，各命令セットについて評価を行う．なお，各DAGの初期推定下限値L_0は，4DCTが8，WEFが22，8DCTが23である．

表2にDAGカバーリングの最適コストを示す．この結果はDAG分割を利用せずに得られる．#Mは利用可能なマッチ数，#Sは最短実行ステップ数を示す．基本分枝限定法の#Sは真の最適解を示す．'NA'は実用的な時間内に計算が完了しないことを意味する．表2に並列マッチ削減のための定数パラメータC=2とした結果を示す．提案する並列マッチ削減が，解の最適性を損なわずに最適DAGカバーリングの発見に要するCPU時間を短縮できることがわかる．

表3にDAG分割の効果を示す．#Mはオリジナル(非分割)DAGのマッチ数を示す．本実験におけるDAG分割には，各DAGについて，DAGを2分割する異なる10パターンを用いる．なお，各分割パターンについて，CPU時間が90分を超えた時点で探索は強制終了する．DAG分割によって最適DAGカバーリングを実用的な時間内に得ることができることを示している．

提案する手法により，与えられたDAGに対して命令セット3と4の方が命令セット2よりも適していること，および命令セット4が与えられたすべてのDAGに対する最適命令セットの1つであるという結果が，実用的な時間内で得られた．

8. 結論

本論文では，与えられたアプリケーション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た．この手法は専用プロセッサの設計フローへ組み込むことができる．

探索処理の更なる高速化，DAG分割の的確な選択，およびレジスタ数制約の考慮が今後の研究課題である．

文 献

[1] B. Shackleford, et al., "Satsuki: An Integrated Processor Synthesis and Compiler Generation System," IEICE Trans. Inf. & Syst., vol.E79-D, No.10, pp. 1373-1381, 1996.

[2] M. Itoh, et al., "Processor Generation Method for Pipelined Processors in Consideration with Pipeline Hazards," IPSJ Journal, vol.41, No.4, pp.851-862, 2000.

[3] T. Sasaki, et all., "Rapid Prototyping of Complex Instructions for Embedded Processors Using PEAS-III," Proc. SASIMI 2000, pp.71-78, 2000.

[4] N. Ishiura, T. Watanabe, and M. Yamaguchi, "A Code Generation Method for Datapath Oriented Application Specific Processor Design," Proc. SASIMI 2000, pp.71-78, 2000.

[5] O. Wahlen, et al., "Instruction Scheduler Generation for Retargetable Compilation," IEEE Design & Test, vol.20, No.1, pp.34-41, 2003.

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

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

[8] Sonia M. Heemstra de Groot et al., "Range-Chart-Guided Iterative Data-Flow Graph Scheduling," IEEE Trans. Circuits Syst.-I: Fund. Theory & Appl., vol. CAS-39, No.5, pp.351-364, 1992.

[9] C. Loeffler et al., "Practical Fast 1-D DCT Algorithms with 11 Multiplications," Proc. IEEE ICASSP, pp.988-991, 1989.