並べ替え命令を考慮したSIMD命令を有するプロセッサのためのコード最適化手法

田中浩明[†1]　武内良典[†1]
坂主圭史[†1]　今井正治[†1]

近年，SIMD 命令は組み込みプロセッサで実装され，マルチメディア向けアプリケーション処理において広く用いられている．SIMD 命令は 1 レジスタ内に詰められた複数個のデータに対して同一の演算を実行する．一方，並べ替え命令は，異なる 2 つのレジスタそれぞれからデータを取り出し，1 レジスタ内にデータを並べ替えて格納する．並べ替え命令を適切に使用することで演算の並列性を活用してより効率的に SIMD 命令を利用できる．しかし，SIMD 命令や並べ替え命令を効率的に利用するコンパイラのコード生成技術は，まだ発展途上である．本稿では，2 並列の SIMD 命令を対象とし，並べ替え命令を活用して SIMD 命令をより効果的に利用するコード生成手法を提案する．提案手法では，並べ替え命令を扱えるようにコード選択問題を拡張する．コード選択問題は整数線形計画問題に定式化して解く．並べ替え命令を扱うことによって，最適なデータの並べ替えを行って SIMD 命令を効率的に利用するコードが生成される．評価実験では，提案手法は並べ替え命令を利用しない場合と比較して，最適化対象のベーシックブロックにおいて，生成するアセンブリコード行数をおよそ 30%，実行サイクル数を 20%削減することが確認された．

A Code Optimization Technique for Processors with SIMD Instructions Considering Permutation Instructions

Hiroaki Tanaka,[†1] Yoshinori Takeuchi,[†1] Keishi Sakanushi[†1]
and Masaharu Imai[†1]

Recently, SIMD insturctions are often implemented in embedded proccessors and widely used for processing multimedia applications. SIMD instructions perform same operations on each data packed in one register. On the other hand, permutation instructions take some data from two registers and, permutate and store them into one register. Using permutation instructions to increase the parallelism adequately, high performance can be lead by exploiting SIMD instructions. However, the code generation methodology to make the best use of SIMD insturctions has not been seen established in compilers. This paper proposes a code selection method for SIMD instructions with permutation instructions. In the proposed method, the code selection problem is extended to handle permutation instructions. Additional nodes are added DAGs which represent data flow in a basic block, then permutation instructions are assigned to the additional nodes. The code selection problems are formulated into integer programing problem and solved by IP solver, so the optimal assembly code exploiting SIMD insturctions can be obtained. Experimental results show that the proposed method reduced the code size by 20.5% and the execution cycles by 23.6% or more, comparing to the method without permutation instructions.

1. はじめに

近年，音声データや画像データを扱うディジタル信号処理アプリケーションがさまざまな電子システム上で利用されている．ディジタル信号処理アプリケーションを実行するための命令セットプロセッサでは，高性能や高設計品質を達成するために，アプリケーションを効率的に実行できる命令セットを持つことが多い．ディジタル信号処理アプリケーションでは，プロセッサのワード長よりも短いビット長のデータに対する同種の演算が多く現れる．この性質を利用して処理を効率良く行うための命令に SIMD（Single Instruction Multiple Data）命令がある．SIMD 命令は，1 レジスタ内の複数のデータに対して並列に同種の演算を実行する命令である．SIMD 命令は複数の演算を並列実

†1 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

行するため，適切に利用することで高い処理性能を達成することができる．また，SIMD 命令は並列実行するための複数個の演算器を組み込むのみで実装でき，VLIW プロセッサやスーパスカラプロセッサなどの命令レベルの並列性を利用するプロセッサと比較して制御機構やデータパスのハードウェア面積は低く，ハードウェアコストに対する性能比は高いという利点がある．このため，近年のディジタル信号処理向けプロセッサ，および，汎用プロセッサの多くで SIMD 命令が実装されている．

しかしながら，SIMD 命令を持つプロセッサ上で実行するプログラムの開発には問題点がある．通常，アプリケーションプログラムの開発では，高級言語でプログラムを記述してコンパイラによってアセンブリコードを生成するが，現在のコンパイラでは SIMD 命令を十分に有効利用するアセンブリコードを生成することはできない．したがって，SIMD 命令をアプリケーションプログラムで利用するには，アセンブリ言語でのプログラムの開発や高級言語から直接的にプロセッサの特定の命令を実行する組み込み関数の利用[1),2)]，また，あらかじめ特定用途向けの命令によって最適化されたライブラリの利用[5)] などの方法をとる必要があるが，開発されるプログラムは特定のプロセッサに依存するためプログラムの移植性は低下する．また，アセンブリ言語によるプログラミングや，プロセッサの命令を直接的に呼び出す組み込み関数を使用するプログラミングは，高級言語でのプログラミングよりもより難しくなるため，アプリケーションの開発工数を増加させる．したがって，プログラムの移植性の低下と開発工数の増加の問題を解決するために，高級言語のプログラムから SIMD 命令を自動的に利用する手法が求められている．

SIMD 命令利用においては，並べ替え命令によるデータの並べ替えによって SIMD 命令を利用する機会をいかに増大させることができるかが重要である．並べ替え命令は，2 つのレジスタからデータを抽出して 1 つのレジスタに並べ替える命令である．一般的に，2 オペランドの SIMD 命令では，異なる 2 つのレジスタ内で同位置にあるデータどうしを演算する．このため，演算対象のデータ群を 2 つのレジスタ内に格納し，演算するデータどうしを異なるレジスタ内の同位置に配置しなければ，SIMD 命令を有効に利用できない．このようなレジスタ内のデータの配置を調整し，SIMD 命令で演算を実行するために並べ替え命令が用意されている．並べ替え命令により，SIMD 命令をより多くの機会で活用することができる．

Leupers はコンパイラのコード選択のフェーズで SIMD 命令を考慮したコード選択手法を提案している[6),7)]．Leupers の手法では，SIMD 命令の要素の演算を抽出した後，SIMD 命令で実行する演算群を決定する問題を整数線形計画問題に帰着させて問題を解き，コードを生成する．しかし，Leupers の手法では並べ替え命令は考慮していない．Eichenberger らは，プログラム中のループ内の配列に対する演算のうち並列化可能な演算を SIMD 命令に割り当てる手法を提案している[9)]．Eichenberger らの手法では，レジスタ内のデータの位置をシフトするコードを挿入して，レジスタ内のデータの位置を調整し SIMD 命令を利用可能にする．レジスタ内のデータ位置のシフトはデータの並べ替えの操作の一種であり，これにより SIMD 命令をより活用できるがデータの順序の入れ換えなどの複雑なデータの並べ替えは考慮していない．Larsen らは，ソースコードの文の列に対して，同時に実行可能な演算を逐次的にグループ化して SIMD 命令に割り当てる手法を提案している[8)]．しかし，Larsen の手法では，データの並べ替えのコストについては深く考察されていない．

Kudriavtsev はレジスタ内のデータの並べ替えも考慮して SIMD 命令を利用する手法を提案している[10)]．並列実行可能な演算を抽出し，演算を SIMD 命令に割り当て，演算を SIMD 命令に割り当てた結果に従ってデータの並べ替え命令列を生成している．Kudriavtsev らの手法は，SIMD 命令およびデータの並べ替え命令列の生成は発見的手法で行っており，生成されるコードの最適性は保証されていない．

本稿では，データの並べ替えも考慮に入れて SIMD 命令を利用する最適化手法を提案する．提案手法では，プログラムを実行する命令を決定する問題であるコード選択問題を整数線形計画問題に定式化して問題を解く．問題の定式化において，データの並べ替えも考慮した制約式を採用する．演算命令間では，レジスタ間でデータを移動しない場合や，並べ替え命令によってレジスタの中のある位置から異なる位置へ移動する場合などが考えられ，それらを制約式で表現することで，データの並べ替えも含めて最適化されたアセンブリコードが得られる．

以下，2 章で SIMD 命令について述べ，3 章で従来手法を述べる．4 章で提案手法を述べ，5 章で実験結果を示し，6 章でまとめる．

2. SIMD 命令

本章では，SIMD 命令，および本稿で着目する並べ

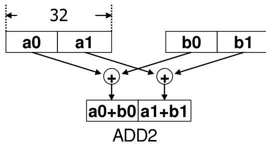


図 1 SIMD 命令
Fig. 1 SIMD instructions.

替え命令について説明し，また，SIMD 命令の利用における問題について説明する．

2.1 SIMD 命令

SIMD（Single Instruction Multiple Data）命令は，レジスタ内の複数のデータに対して，1 命令で同種の複数の演算を同時に行う命令である．SIMD 命令は複数個の演算を同時に行うため，演算に必要な命令数，実行サイクル数を削減することができる．一般的に，加減算や乗算，シフト演算などの SIMD 命令が使用され，メモリの読み書きでは，ワード単位のメモリアクセス命令が SIMD のロード/ストア命令として用いられる．近年では信号処理アプリケーションの利用機会が多いため，汎用プロセッサ[1),2)]，信号処理用途のプロセッサ[3),4)] の多くが SIMD 命令を持つ．**図 1** に 2 並列 SIMD 加算の命令の例を示す．図 1 では，32 ビットレジスタに 16 ビットデータが 2 個入っており，2 つのレジスタで同位置にあるデータに対してそれぞれ加算を行い，結果を 1 つのレジスタに格納する．

2.2 並べ替え命令

SIMD 命令を持つプロセッサの多くは，レジスタ内のデータの並べ替えや詰め替えを行う並べ替え命令も実装されている．並べ替え命令は，演算対象データの配置がレジスタ内で SIMD 命令を適用できない配置になっている場合に，データを並べ替えて SIMD 命令を適用するために用いる．

本稿では，以下の特徴を持つ命令を並べ替え命令と呼ぶ．

- 2 つのレジスタを入力オペランドにとる．
- 1 つのレジスタを出力オペランドにとる．
- 入力オペランドのレジスタは，同一ビット数の複数のデータを内部に保持する．
- 入力オペランドのレジスタから，複数のデータを取り出し，それらを 1 つのレジスタに格納する．取り出すデータのレジスタ内の位置と格納先のレジスタ内の位置は任意とする．

図 2 にテキサスインスツルメンツ社の信号処理向けプロセッサ TMS320C62xx[3)] が持つ並べ替え命令の例を示す．図 2 の並べ替え命令は，2 つのデータが格納されているレジスタ 2 つをオペランドにとり，そ

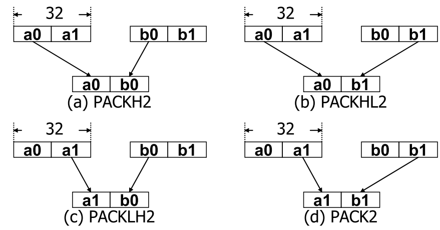


図 2 並べ替え命令
Fig. 2 Permutation instructions.

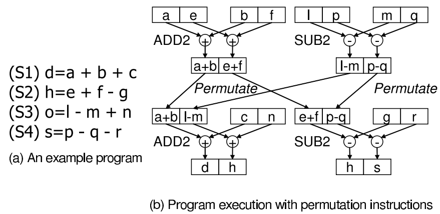


図 3 並べ替え命令の利用例
Fig. 3 An example of permutation instruction utilization.

れぞれのレジスタからデータを 1 つ取り出し，1 つのレジスタに格納する．図 2 (a) は，2 つのレジスタの上位から，図 2 (b)，(c) は上位と下位から，図 2 (d) は下位からそれぞれデータを取り出す．

2.3 SIMD 命令利用における問題

SIMD 命令を用いる場合，2 つの演算対象のデータはそれぞれのレジスタ内で同位置に配置されていなければならない．**図 3** (a) は 4 つの文があり，S1 は加算の次に加算，S2 は加算の次に減算，S3，S4 はそれぞれ減算の次に加算，減算の次に減算となっている．図 3 (a) の文はそれぞれ互いに異なるため並べ替え命令なしでは SIMD 命令で演算を実行できない．並べ替え命令なしの場合，図 3 (a) では 8 個の加減算があるため，1 演算を 1 命令で実行すると合計で 8 命令必要である．しかし，SIMD 命令と並べ替え命令を用いると，ADD2 を 2 回，SUB2 を 2 回，並べ替え命令を 2 回の 6 命令で演算を実現できる．図 3 (a) の演算を 2 並列 SIMD 加算命令 ADD2，2 並列 SIMD 減算 SUB2，並べ替え命令を用いた実行例を，図 3 (b) に示す．図 3 のように，並べ替え命令と SIMD 命令の併用により，全体の演算を実行する命令数を削減できる．

SIMD 命令を利用する場合には演算対象データのレジスタ内の位置が一致している必要があるが，図 3 に示すように，演算対象データの位置がレジスタ内で一致しない場合でも並べ替え命令を用いてレジスタ内の位置を一致させることで，SIMD 命令を適用できる．しかしながら，レジスタ内のデータの位置の調整も考慮したコード生成手法は確立しておらず，コンパイラ

が SIMD 命令を有効に利用できる機会は少ない．

3. 従来の SIMD 命令最適化

本章では，整数線形計画法による SIMD 命令最適化[6] について説明する．Leupers は，コンパイラのコード選択において，SIMD 命令を含む命令を選択できるようにコード選択問題を定式化し，問題を解いている．以降，コード選択について説明し，次に，Leupers の手法を説明する．

3.1 コード選択

コード選択は，コンパイラのコード生成において，プログラムを実行する命令列を決定する手続きである．コード選択では，プログラムの処理をデータフローグラフ（Data Flow Graph，DFG）で表現し，木文法で定義された規則による DFG の節点の被覆によって命令を決定する．コード選択はベーシックブロックごとに適用される．ベーシックブロック内の処理を，演算を節点，辺をデータの依存関係を表す DFG で表現する．さらに，DFG を出力辺が 2 以上の節点で分割して，ベーシックブロック内の処理をデータフローツリー（Data Flow Tree，DFT）の集合で表現する．

木文法は，非終端記号の集合，終端記号の集合，規則の集合，開始記号，コスト関数で構成される．非終端記号はレジスタに対応し，終端記号は演算，定数，メモリに対応する．規則は命令の動作を表現する．たとえば，2 つのレジスタの値の和をレジスタに代入する ADD 命令は次のように表す．

$$reg \rightarrow PLUS(reg, reg) \quad (1)$$

ここで，reg は非終端記号であり，$PLUS$ は終端記号である．コード選択では，DFT を最小コストで被覆する木文法の規則の集合を求める．木文法の各規則はコスト関数によって重みづけし，導出過程で適用するすべての規則のコストの和をその導出木のコストとする．コスト関数は実際の命令に応じて命令の実行サイクル数などを設定する．**図 4** に木文法と DFT の木文法による被覆の例を示す．図 4 (a) は木文法であり，メモリアクセス（MEM），加算（PLUS），乗算（MULT）を表す終端記号，汎用レジスタ（reg）を表す非終端記号，4 つの命令を表す規則を持ち，開始記号が reg である．図 4 (b) は DFT と，DFT を被覆する規則を点線で示している．図 4 (b) で，MEM の節点は規則 LD で被覆され，図 4 (b) 左の乗算の点は MUL で，図 4 (b) 右の乗算と加算の点は MAC で被覆されている．

3.2 コード選択における SIMD 命令の導入

Leupers は，木文法に SIMD 命令を表現する規則を

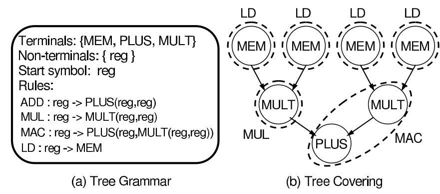


図 4　木文法と規則による被覆
Fig. 4　Tree grammer and covering by rules.

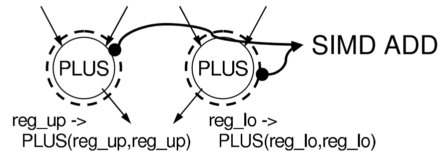


図 5　SIMD 命令の規則による被覆
Fig. 5　Covering by SIMD rules.

導入し，SIMD 命令利用を最適化する問題に定式化している．DFG の各節点では，その演算を実行する命令について，通常の命令による実行，または，SIMD 命令による実行の選択肢がある．さらに，SIMD 命令を利用する場合には，レジスタ内の位置を考慮して適用する規則を決定しなければならない．たとえば，2 並列の SIMD 加算命令を持つプロセッサにおいて加算を行う場合，以下の選択肢が考えられる．

- 通常の命令で実行
- SIMD 命令で実行（レジスタの上位）
- SIMD 命令で実行（レジスタの下位）

木文法では，上記の選択肢をそれぞれ以下の規則で記述する．

1:　$reg \rightarrow PLUS(reg, reg)$

2:　$reg_up \rightarrow PLUS(reg_up, reg_up)$

3:　$reg_lo \rightarrow PLUS(reg_lo, reg_lo)$

reg，reg_up，reg_lo はそれぞれレジスタ全体，レジスタの上位，レジスタの下位を表す．また，$PLUS$ は加算を表す．命令を決定する際，通常の加算は規則 1 の被覆で決定され，SIMD 命令の加算は規則 2，3 の被覆を 1 命令とすることで決定される．**図 5** に，SIMD 命令の規則による被覆の概念図を示す．規則 2，3 に被覆された節点を，SIMD 加算の 1 命令に割り当てる．SIMD 命令の規則による被覆では，SIMD 命令の複数演算同時実行制約と命令の無矛盾な依存の，SIMD 命令導入による制約を満たす必要がある．Leupers は，これらの制約を整数線形計画問題に定式化し，SIMD 命令を生成している．しかしながら，Leupers の定式化ではデータの並べ替えを考慮していないため，データの並べ替えによって SIMD 命令が利用できる場合でも SIMD 命令を利用するコードを生成できない．

4. 並べ替え命令を考慮した SIMD 命令最適化

本章では，並べ替え命令を考慮した SIMD 命令最適化手法を提案する．提案手法では，並べ替え命令を導入して並べ替え命令も利用して最適化をするように問題を定式化する．提案手法では，コード選択で並べ替え命令の選択を可能にするため，データ移動節点を追加した DFG である，DFG-DM（Data Flow Graph with Data Move Node）を導入する．木文法の規則で並べ替え命令を表現する規則を導入する．そして，コード選択問題を並べ替え命令の選択も可能にした整数線形計画問題を定式化する．本章では，ワード長が 32 ビット，SIMD 命令の並列度数が 2，すべての可能なデータの並べ替えを実行可能な命令セットを持つ場合を仮定している．

4.1 データ移動節点の追加

コード選択では DFG の各節点を規則で被覆する．DFG では演算のみが表現されているため，並べ替え命令を適用する箇所を見つけるのは困難である．そこでデータの演算を表す節点の間にデータの移動を表現する節点を導入する．データ移動を表現する節点を追加して得られる DFG を DFG-DM と呼ぶ．

DFG-DM は，DFG のすべての辺について，その辺の両端点の間に新たにデータの移動を表現する節点を挿入して得られるグラフとする．DFG-DM は，DFG のすべての節点を訪問し，節点が出力辺を持つ場合に出力辺の接続先の点との間にデータの移動を表現する節点を挿入して，既存の節点と新たに挿入した節点を辺で接続することで生成する．

図 6 (a) の DFG にデータ移動節点を追加して得られる DFG-DM を図 6 (b) に示す．三角の節点がデータ移動節点である．これらの節点はデータ移動のための規則で被覆する．

4.2 コード選択における並べ替え命令の導入

提案手法では，コード選択において並べ替え命令を導入する．データの移動についても，SIMD 命令の利用と同様，並べ替え命令によるレジスタ内のデータの位置の移動，または並べ替え命令が必要ない場合の選択肢がある．たとえば，2 並列の並べ替え命令の場合，以下の選択肢がある．

- レジスタの下位にあるデータを下位に移動する．
- レジスタの上位にあるデータを下位に移動する．
- レジスタの下位にあるデータを上位に移動する．
- レジスタの上位にあるデータを上位に移動する．
- レジスタの上位にあるデータを移動しない．

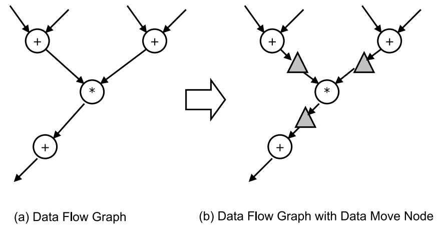
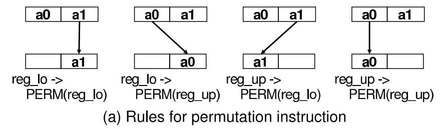
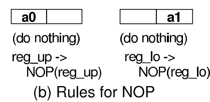


図 6 データ移動節点の追加
Fig. 6 Insertion of move nodes.





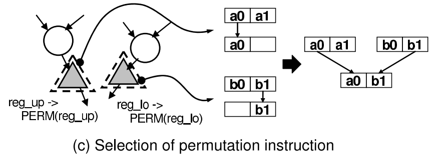


図 7 並べ替え命令の規則と選択
Fig. 7 Rules for permutation instructions and selection of the rules.

- レジスタに下位にあるデータを移動しない．

上記の選択肢で最後の 2 つの選択肢は，並べ替え命令が必要ない場合の選択肢である．Leupers の手法と同様に，上記の選択肢を以下の規則で表現する．

1: $reg_lo \rightarrow PERM(reg_lo)$
2: $reg_lo \rightarrow PERM(reg_up)$
3: $reg_up \rightarrow PERM(reg_lo)$
4: $reg_up \rightarrow PERM(reg_up)$
5: $reg_lo \rightarrow NOP(reg_lo)$
6: $reg_up \rightarrow NOP(reg_up)$

コード選択で命令を決定する際，並べ替え命令は，規則 1–4 の任意の 2 つの被覆の組合せを 1 命令とすることで決定される．データを並べ替えない場合は，規則 5，6 による被覆で決定される．**図 7** に，並べ替え命令の規則と動作との対応，および，並べ替え命令の選択例を示す．図 7 (a) はデータの移動の規則を，図 7 (b) は，データを移動しないことを示す規則を表している．図 7 (c) は，並べ替え命令の選択例であり，規則 4 と規則 1 が選択される場合，結果として生成される並べ替え命令は，2 つのレジスタからそれぞれ上位と下位のデータをとって 1 レジスタに格納する命令となることを表している．

4.3 整数線形計画問題の定式化

ベーシックブロックのデータフローを表す DFG と，命令を表現する規則の集合に対して DFG の各節点に対して被覆する規則の候補を列挙する．$V = \{v_1, ..., v_n\}$ を DFG-DM の節点の集合とする．r_{ik} を v_i を被覆する規則の集合 $R(v_i)$ の中の規則の 1 つとする．

4.3.1 変数の定義

変数 x_{ik} は，整数線形計画問題（ILP）を解いたときに，それぞれの節点で選択された規則を表し，次のように定義する．

$$x_{ik} = \begin{cases} 1, & v_i \text{ を規則 } r_{ik} \text{ で被覆する} \\ 0, & v_i \text{ を規則 } r_{ik} \text{ で被覆しない} \end{cases} \quad (2)$$

次の条件を満たす 2 節点は，1 つの SIMD 命令で被覆される節点の組の候補である．

- v_i と v_j は並列実行可能である．すなわち，DFG 上で v_i から v_j への経路，v_j から v_i への経路がない．
- v_i と v_j はハーフワードデータに対する同じ演算である．v_i と v_j が定数の場合，それぞれハーフワードの定数でなければならない．
- $R(v_i)$ の中に，規則の左辺が reg_up であるものがあり，$R(v_j)$ の中に，規則の左辺が reg_lo であるものがあること．
- v_i と v_j がともにメモリアクセスの演算である場合は，それぞれの参照するメモリアドレスが連続していること．

以上の条件を満たす節点の組 (v_i, v_j) の集合を P とする．$(v_i, v_j) \in P$ に対して変数 y_{ij} を定義する．y_{ij} が 1 のとき，v_i が上位のレジスタで実行され，v_j が下位のレジスタで実行されることを意味する．

$$y_{ij} = \begin{cases} 1, & v_i \text{ と } v_j \text{ を SIMD 命令で実行する} \\ 0, & v_i \text{ と } v_j \text{ を SIMD 命令で実行しない} \end{cases} \quad (3)$$

$V_{op} \subset V$ を演算を表現する節点の集合，$V_{move} \subset V$ を移動表現の節点の集合とし，$P_{move} = \{(v_i, v_j) \in P | v_i, v_j \in V_{move}\}$ とする．$(v_i, v_j) \in P_{move}$ である v_i，v_j に対して変数 a_{ij}，b_{ij} を定義する．a_{ij} は，v_i と v_j を並べ替え命令を用いて 1 レジスタにデータを移動することを示す．b_{ij} は，v_i と v_j では並べ替え命令を用いずデータの移動は行わないこと，さらに，データの移動を行わなくても，v_i と v_j が 1 レジスタ内に存在していること，すなわち，SIMD 命令の実行によって得られた結果であることを表す．

$$a_{ij} = \begin{cases} 1, & v_i \text{ と } v_j \text{ を並べ替え命令で 1 レジスタに移動} \\ 0, & \text{その他} \end{cases}$$

$$b_{ij} = \begin{cases} 1, & v_i \text{ と } v_j \text{ は 1 レジスタ内に存在し移動しない} \\ 0, & \text{その他} \end{cases}$$

4.3.2 制約の定義

本項では，制約式について述べる．制約式について述べる前に，制約式で使用する記号を定義する．

v_i を被覆する規則 $r_{ik} \in R(v_i)$ において r_{ik} で出現する非終端記号を出現順に $nt_{ik1}, nt_{ik2}, .., nt_{ikm}, ..$ とする．また，$R(v_i)$ の要素で，左辺の非終端記号が nt である規則の集合を $R^{nt}(v_i)$ とする．また，$R^{nt}_{simd}(v_i)$ を，$R(v_i)$ の要素で，左辺の非終端記号が nt であり SIMD 命令の規則である集合と定義し，$R^{nt}_{perm}(v_i)$，$R^{nt}_{nop}(v_i)$ も同様に定義する．

上記の定義の例を示す．節点 v_i を被覆する規則が以下の規則の集合 $R(v_i) = \{r_{i1}, r_{i2}, r_{i3}\}$ であるとする．

$$\begin{aligned} r_{i1} &: reg \rightarrow PLUS(reg, reg) \\ r_{i2} &: reg_up \rightarrow PLUS(reg_up, reg_lo) \\ r_{i3} &: reg_lo \rightarrow PLUS(reg_lo, reg_up) \end{aligned}$$

このとき，r_{i2} で出現する非終端記号は，左から reg_up，reg_up，reg_lo である．したがって，$nt_{i21} = reg_up$，$nt_{i22} = reg_up$，$nt_{i23} = reg_lo$ である．同様に，r_{i3} では，$nt_{i31} = reg_lo$，$nt_{i32} = reg_lo$，$nt_{i33} = reg_up$ となる．また，$R(v_i)$ の中で SIMD 命令の規則は r_{i2} と r_{i3} であるので，$R^{nt}_{simd}(v_i)$ で $nt = reg_up$ とすると $R^{reg_up}_{simd}(v_i) = \{r_{i2}\}$ となり，$nt = reg_lo$ とすると，$R^{reg_lo}_{simd}(v_i) = \{r_{i3}\}$ となる．

また，移動表現の節点 $v_{i'}$ において，$v_{i'}$ を被覆する規則が以下の規則の集合 $R(v_{i'}) = \{r_{i'1}, r_{i'2}, .., r_{i'6}\}$ であるとする．

$$\begin{aligned} r_{i'1} &: reg_lo \rightarrow PERM(reg_lo) \\ r_{i'2} &: reg_lo \rightarrow PERM(reg_up) \\ r_{i'3} &: reg_up \rightarrow PERM(reg_lo) \\ r_{i'4} &: reg_up \rightarrow PERM(reg_up) \\ r_{i'5} &: reg_lo \rightarrow NOP(reg_lo) \\ r_{i'6} &: reg_up \rightarrow NOP(reg_up) \end{aligned}$$

このとき，$R^{reg_lo}_{perm}(v_{i'}) = \{r_{i'1}, r_{i'2}\}$，$R^{reg_up}_{perm}(v_{i'}) = \{r_{i'3}, r_{i'4}\}$，$R^{reg_lo}_{nop}(v_{i'}) = \{r_{i'5}\}$，$R^{reg_up}_{nop}(v_{i'}) = \{r_{i'6}\}$ となる．

以下，制約式を説明する．

- 規則選択の排他制約

各節点は 1 つの規則を選択する．

$$\forall v_i : \sum_{r_{ik} \in R(v_i)} x_{ik} = 1 \quad (4)$$

- 非終端記号の一致制約
  v_i の親の節点の集合を $A(v_i)$ とする．このとき，v_i と $v_{i'} \in A(v_i)$ で選択される規則は，規則の導出で用いられる非終端記号が一致していなければならない．この制約は次の式で表される．

$$\forall v_i \in V : \forall r_{ik} \in R(v_i) : \forall m > 1 :$$
$$x_{ik} \leq \sum_{\substack{v_{i'} \in A(v_i), r_{i'k'} \in R(v_{i'}), \\ nt_{i'k'm} = nt_{ik1}}} x_{i'k'} \quad (5)$$

- SIMD 命令における同時演算制約
  もし，$r_{ik} \in R^{reg_up}_{simd}(v_i)$ が v_i で選択されたら，$(v_i, v_j) \in P$ であるような v_j もまた，被覆されなければならない．これは，$y_{ij} = 1$ で表現される．この制約は次のように表せる．

$$\forall v_i \in V_{op} :$$
$$\sum_{r_{ik} \in R^{reg_up}_{simd}(v_i)} x_{ik} = \sum_{(v_i, v_j) \in P} y_{ij} \quad (6)$$
$$\forall v_j \in V_{op} :$$
$$\sum_{r_{jk} \in R^{reg_lo}_{simd}(v_j)} x_{jk} = \sum_{(v_i, v_j) \in P} y_{ij} \quad (7)$$

- 演算の実行順序制約
  節点 v_i について，$X(v_i)$ を v_i より先に実行しなければならない節点の集合，$Y(v_i)$ を v_i より後に実行しなければならない節点の集合とする．$(v_i, v_j) \in P$ が SIMD 命令によって被覆されるとしたら，次の集合 Z_{ij} に含まれる点の組を SIMD 命令で実行することはできない．

$$Z_{ij} = \{(v_p, v_q) \in P |$$
$$v_p \in X(v_i) \wedge v_q \in Y(v_j)$$
$$\vee v_p \in X(v_j) \wedge v_q \in Y(v_i)\} \quad (8)$$

  (v_i, v_j) と Z_{ij} の要素の排他的選択は，次の式で表せる．

$$\forall (v_i, v_j) \in P : \forall (v_p, v_q) \in Z_{ij} : y_{ij} + y_{pq} \leq 1 \quad (9)$$

- レジスタ間データ移動における同時実行演算の存在制約
  レジスタ間のデータ移動では，並べ替え命令によってデータの移動を行うか，あるいはデータの移動を行わないかが選択肢として存在する．それぞれの場合での制約は以下のとおりである．
  - 並べ替え命令は，2 つのデータの移動の組で実行されなければならない．このため，ある節点の規則に PERM が選ばれるときは，必ず組となる節点を必要とし，その節点もまた PERM が選ばれなければならない．
  - NOP が組で実行されるときは組となる節点も NOP が選ばれなければならない．

  以上の制約の式は次のように表現できる．

$$\forall v_i \in V_{move} :$$
$$\sum_{r_{ik} \in R^{reg_up}_{perm}(v_i)} x_{ik} = \sum_{(v_i, v_j) \in P_{move}} a_{ij} \quad (10)$$
$$\forall v_j \in V_{move} :$$
$$\sum_{r_{jk} \in R^{reg_lo}_{perm}(v_j)} x_{jk} = \sum_{(v_i, v_j) \in P_{move}} a_{ij} \quad (11)$$
$$\forall v_i \in V_{move} :$$
$$\sum_{r_{ik} \in R^{reg_up}_{nop}(v_i)} x_{ik} \geq \sum_{(v_i, v_j) \in P_{move}} b_{ij} \quad (12)$$
$$\forall v_j \in V_{move} :$$
$$\sum_{r_{jk} \in R^{reg_lo}_{nop}(v_j)} x_{jk} \geq \sum_{(v_i, v_j) \in P_{move}} b_{ij} \quad (13)$$

  また，y_{ij} は 2 つの節点 v_i と v_j を並列実行するときに 1 となる．この定義と a_{ij}，b_{ij} の定義より，次の制約が必要となる．

$$\forall (v_i, v_j) \in P_{move} : y_{ij} = a_{ij} + b_{ij} \quad (14)$$

- SIMD 命令の被演算データレジスタの同一性制約
  (v_i, v_j) が SIMD 命令で実行できるためには，v_i と v_j で演算対象のデータがレジスタの中で SIMD 命令を実行できる位置に配置されていなければならない．v_i の左右の入力節点，v_j の左右の入力節点をそれぞれ v_{i_l}，v_{i_r}，v_{j_l}，v_{j_r} とする．v_i と v_j が SIMD 命令で実行されるとすると，v_{i_l} と v_{j_l}，v_{i_r} と v_{j_r} の演算結果がそれぞれ 1 つのレジスタ内に配置されていなければならない．これは，それぞれの節点もまた SIMD 命令で実行されたことを意味している．**図 8** に例を示す．図 8 (a) の DFG を SIMD 命令で実行する際の入出力のレジスタ内のデータの関係を図 8 (b) に示している．図 8 (a) で v_i，v_j，v_{i_l}，v_{i_r}，v_{j_l}，v_{j_r} の演算結果のデータをそれぞれ d_i，d_j，d_{i_l}，d_{i_r}，d_{j_l}，d_{j_r} とし，v_i，v_j が SIMD 命令で実行されるとすると，d_{i_l} と d_{j_l}，d_{i_r} と d_{j_r} はそれぞれ 1 つのレジスタ内に存在しなければならない．これは，d_{i_l} と d_{j_l}，d_{i_r} と d_{j_r} が SIMD 命令または並べ替え命令の実行結果であるということを意味する．
  v_i と v_j，v_{i_l} と v_{j_l}，v_{i_r} と v_{j_r} の SIMD 命令による実行を表現する変数は y_{ij}，$y_{i_l j_l}$，$y_{i_r j_r}$ であり，上述の制約は y_{ij} が 1 ならば，$y_{i_l j_l}$，$y_{i_r j_r}$ がともに 1 でなければならないことを意味してい

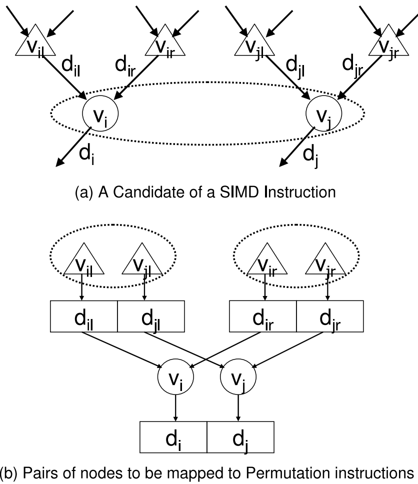


図 8　SIMD 命令の被演算データレジスタの同一性
Fig. 8　Identity of operand registers of SIMD instructions.

る．この制約は以下の式で表現できる．

$$y_{ij} \leq y_{i_l j_l} \tag{15}$$

$$y_{ij} \leq y_{i_r j_r} \tag{16}$$

4.3.3　目 的 関 数

本手法の目的は命令数の最小化であり，目的関数は式 (17) のようになる．式 (17) において，第 1 項は，通常の命令が適用された，算術，論理演算，ロード，ストアを表現する節点での命令数である．第 2 項は，SIMD 命令が選択された数である．第 3 項は，並べ替え命令が選択された数である．

$$\begin{aligned} &minimize \sum_{v_i \in V - V_{move}} \sum_{r_{ik} \in R^{reg}(v_i)} x_{ik} \\ &\quad + \sum_{(v_i, v_j) \in P - P_{move}} y_{ij} + \sum_{(v_i, v_j) \in P_{move}} a_{ij} \end{aligned} \tag{17}$$

5.　評 価 実 験

評価実験では，提案手法が並べ替え命令を利用して SIMD 命令を有効に活用するコードが生成できることを確認するために，DLX [11] プロセッサと DSPstone [12] ベンチマークプログラムを用い実験を行った．DLX プロセッサに対して図 1 に示した 1 命令で 2 つの加算を行う ADD2 命令，および，1 命令で 2 つの乗算を行う MULT2 命令，図 2 に示した 4 種類の並べ替え命令，PACKH2，PACKHL2，PACKLH2，PACKM2 を追加したプロセッサを用いた．DLX プロセッサは，5 段パイプラインの RISC アーキテクチャ，同時発行命令数は 1 命令であり，整数演算命令のみを持つ．演算命令のレイテンシはすべて 1 としており，乗除算命令以外の命令はすべて 1 サイクル，乗算命令，除算命令はそれぞれ 32 サイクル，35 サイクルとしている．

本実験では，DSPstone ベンチマークプログラムの中から 6 種類のプログラムを，以下の 3 つのコンパイラでコンパイルした．

SIMD なし　SIMD 命令を利用しないコンパイラ
従来法　SIMD 命令を利用し，並べ替え命令を利用しないコンパイラ
提案法　提案手法を実装したコンパイラ

なお，convolution，fir，matrix では基本ブロック内で並列実行できる演算が少ないため，ループを展開して実験を行った．実験は Intel Xeon 2.8 GHz のプロセッサ，メモリ 2 GB を搭載，OS が RedHat Linux 8.0 の計算機上で行った．コンパイラは，ACE 社[13] のコンパイラ開発ツール CoSy を用いて，提案手法および従来法を組み込んだコンパイラを開発した．ILP の解法には，opbdp [14] を用いた．また，提案法の有効性の評価のために，シミュレーションによって実行サイクル数を取得した．実行サイクル数は，評価対象プロセッサの RTL モデルを HDL で設計し，HDL シミュレーションを実行した結果から取得した．なお，実行サイクル数は，プログラムの初期化などを除いた，主要な処理の実行にかかるサイクル数である．

表 1 に，プログラムをコンパイルしたときの命令数と，シミュレーションによってプロセッサ上でプログラムを動作させたときの実行サイクル数を示す．また，**表 2** に SIMD なしに対する従来法と提案法の，命令数および実行サイクル数の削減率を示す．**表 3** に各プログラムに対して，従来法，提案法でコンパイルした場合の，主要な処理の部分の最適化で解いた ILP の変数の数，制約の数，プログラム全体をコンパイルするためにかかった時間を示す．

5.1　コードの効率に関する考察

表 1 より，SIMD なしと従来法を比較すると，提案法は，すべての場合でコードサイズと実行サイクル数が削減できていることが分かる．また，表 2 から，従来法に対する提案法の命令数は 5 つのプログラムで 30%以上，実行サイクル数は 5 つのプログラムで 20%以上改善されていることが分かる．

従来法と提案法を比較すると，従来法は多くの場合でコードサイズ，実行サイクル数の削減は行えていない．これは，従来法ではデータの並べ替えを考慮していないために，レジスタ内の上位のデータと下位のデータに対して演算するコードが生成できないためである．それに対して，提案法では並べ替え命令によってレジスタ内のデータの入れ換えを行って SIMD 命

表 1 命令数，実行サイクル数の比較
Table 1 The comparison of the number of instructions and execution cycles.

プログラム	SIMD なし		従来法		提案法	
	命令数	サイクル数	命令数	サイクル数	命令数	サイクル数
convolution	31	521	31	521	20	337
dot product	23	69	23	69	16	56
fir	49	664	49	664	27	477
matrix	23	31,684	23	31,684	16	25,184
n complex update	48	1,684	48	1,684	28	1,028
n real update	31	516	18	276	18	276

表 2 命令数，実行サイクル数の削減率比較
Table 2 The comparison of the reduction ratio of the number of instruction and execution cycles.

プログラム	従来法		提案法	
	命令数削減率	サイクル数削減率	命令数削減率	サイクル数削減率
convolution	0.0	0.0	35.5	35.3
dot product	0.0	0.0	30.4	18.8
fir	0.0	0.0	44.9	28.2
matrix	0.0	0.0	30.4	20.5
n complex update	0.0	0.0	41.7	39.0
n real update	41.9	46.5	41.9	46.5

表 3 グラフ節点数，ILP の変数の数，制約の数，実行時間の比較
Table 3 The comparison of the number of DFT nodes, variables and constraints in ILP and CPU time.

プログラム	従来法			提案法		
	変数の数	制約の数	実行時間 [sec]	変数の数	制約の数	実行時間 [sec]
convolution	29	59	0.10	556	2,883	0.36
dot product	34	80	0.11	557	2,884	0.32
fir	41	90	0.13	731	4,740	0.65
matrix	34	80	0.15	557	2,884	0.37
n complex update	64	140	0.13	1,916	22,548	15.55
n real update	40	95	0.13	634	3,515	0.47

令を利用しており，結果として多くの場合でコードサイズを削減している．提案法でのみコードサイズを削減した場合について，評価に用いたプログラムの 1 つである n complex updates を例に説明する．実験で用いた n complex updates のプログラムを**図 9** に示す．n complex updates では，演算対象のデータのうち，A[i] と A[i+1]，B[i] と B[i+1] はメモリ上で隣接しており，それぞれ 2 つのデータを 1 レジスタにロードすることができる．しかしながら，プログラム中の A[i+1] * B[i] の演算は，レジスタ内のデータの位置が A[i+1] が下位に，B[i] が上位になるため，データをレジスタにロードした直後に SIMD 命令の MULT2 命令を利用して演算を実行することはできない．提案法では並べ替え命令を用い，MULT2 命令によって乗算するコードを生成できており，コードサイズを削減している．しかしながら，従来法ではデータの並べ替えを考慮していないために，SIMD 命令を利用するコードは生成できていない．

```
short A[2*N], B[2*N], C[2*N], D[2*N];
int i;

for(i=0;i<N;i+=2) {
  D[i]   = C[i]   + A[i]   * B[i]
                  - A[i+1] * B[i+1];
  D[i+1] = C[i+1] + A[i+1] * B[i]
                  + A[i]   * B[i+1];
}
```

図 9 n complex updates のプログラム
Fig. 9 Program of n complex updates.

Leupers [6] の手法ではデータの並べ替えは考慮していないが，その評価では多くの場合でコードサイズが削減されている．これは，評価で用いた命令セットが本実験とは異なり，**図 10** (a) に示すようなレジスタ内

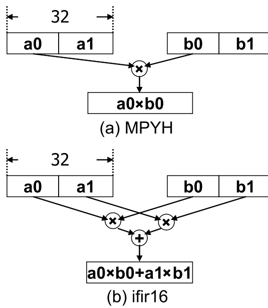


図 10 1 レジスタ内の複数データに対する演算命令
Fig. 10 Data operation instructions on multiple data within a register.

で異なる位置に配置されているデータに対する乗算命令[3] や，図 10 (b) に示すような 1 レジスタ内の複数データに対する演算命令[4] など，SIMD 命令ではないが，1 レジスタ内に複数データを持つデータに対して演算する命令を扱っているためである．これらの命令により，複数データを 1 レジスタにロードした後に，データの並べ替えなしで演算を実行するコードを生成できている．

本実験では，図 10 のような特殊な命令を持たない場合でも SIMD 命令を利用できることを確認するため，そのような命令は含めずに評価を行った．結果として，n complex updates のように並べ替え命令を用いてレジスタ内のデータを並べ替えることで SIMD 命令を利用するコードを生成でき，性能が向上することを確認した．

5.2 コンパイル時間に関する考察

コンパイル時間について考察する．表 2 より，提案法は従来法と比較してコンパイル時間は長い．しかし，多くのプログラムでは実用的な時間で終了しており，また，最適化効果は高いため，コンパイル時間が長くても実用性は高いと考えられる．最も時間のかかっている n complex updates では，変数の数がおよそ 1,900，制約の数がおよそ 22,000 であり，他のプログラムのコンパイルは 1 秒以内で終了しているのに対して 15 秒以上時間がかかっている．n complex update は，他のプログラムと比較して演算数が多く，また，同種の演算が多く含まれており，SIMD 命令での実行の候補数が多くなる．そのために探索する SIMD 命令や並べ替え命令の実行の探索空間が非常に大きくなると考えられる．

また，ILP の実行時間の短縮が課題の 1 つと考えられる．現在の定式化で使用する制約式や変数の中には，すべての制約式を列挙した時点で冗長になるものが含まれる場合があるため，それらを除去して問題を解く方法などが考えられる．

5.3 高並列度の SIMD 命令についての考察

本稿では，SIMD 命令の並列度が 2 の場合のコード最適化手法を提案した．Leupers は，並べ替え命令を考慮しない場合について，より高並列の SIMD 命令のコード最適化手法も提案している[7]．Leupers の方針を用いて，並べ替え命令に対応した高並列度の SIMD 命令最適化が考えられる．

高並列な SIMD 命令，並べ替え命令に対応する方針は，DFG を被覆する規則と ILP の定式化の拡張であり，高並列の SIMD 命令に対しても適用できると考えられる．しかしながら，並列度が高くなると SIMD 命令で実行する演算の組合せ数は大きくなるため，ILP の変数，制約式は並列度に対して指数関数的に増大し，ILP を実用的な時間で解くことは困難であると考えられる．

5.4 並べ替え命令セットの変更に対する考察

本稿では，SIMD 命令の並列度が 2 の場合であり，考えられるすべての並べ替え命令が使用できると仮定していた．しかしながら，プロセッサの命令セットによっては使用できる並べ替え命令が限られる場合がある．使用可能な並べ替え命令が限られる場合には，プロセッサの持つ並べ替え命令セットに従って，制約式を適切に定義することで手法を適用できると考えられる．制約式を定義する方針として，使用できない並べ替えの組合せによる並べ替えの選択を禁止する制約式を追加する方針が考えられる．

たとえば，図 2 (a) の PACKH2 命令は，$reg_hi \rightarrow PERM(reg_hi)$ と $reg_lo \rightarrow PERM(reg_hi)$ の 2 つの規則を選択することで，PACKH2 を使用することが決定されるが，PACKH2 をプロセッサが持たない場合に，$reg_hi \rightarrow PERM(reg_hi)$ と $reg_lo \rightarrow PERM(reg_hi)$ を同時に選択することを禁止する式を制約式として追加する．

上記の方針で適用できると考えられるが，実際の具体的な制約式，目的関数の定義は，今後の課題である．

6. おわりに

本稿では，並べ替え命令を考慮した SIMD 命令のコード選択を行う手法を提案した．コード選択のフェーズにおいて，プログラムの処理を表現するグラフへデータ移動を表現する点の追加，木文法への並べ替え命令の規則の追加を行い，整数線形計画法によるコード選択問題に並べ替え命令を導入して命令数を最小化する問題を解いた．評価実験によって，これらの拡張を行ったコード選択は並べ替え命令を考慮しない SIMD

命令最適化よりも効率の良いコードを生成することを確認した．

今後の課題として，コンパイル時間を短くするための発見的手法の開発や，より高並列な SIMD 命令の最適化があげられる．

参 考 文 献

1) Intel Architecture Optimization Reference Manual (2006). http://developer.intel.com/design/pentiumii/manuals/245127.htm
2) VIS Instruction Set User's Manual (2006). http://www.sun.com/processors/vis/vsdkfiles.html
3) Texas Instruments, TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide (2000).
4) Philips Semiconductors, PNX 1300 Series Databook (2002).
5) Intel Math Kernel Library 9.0 (2006). http://www.intel.com/cd/software/products/asmo-na/eng/307884.htm
6) Leupers, R.: Code Selection for Media Processors with SIMD instructions, *Proc. conference on Design, Automation and Test in Europe* (2000).
7) Leupers, R.: *Code Optimization Techniques for Embedded Processors*, Kluwer Academic Publishers (2000).
8) Larsen, S. and Amarasinghe, S.: Exploiting Superword Level Parallelism with Multimedia Instruction Sets, *ACM SIGPLAN Notices, Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vol.35 Issue 5 (2000).
9) Eichenberger, A.E., Wu, P. and O'Brien, K.: Vectorization for SIMD architectures with alignment constraints, *ACM SIGPLAN Notices, Proc. ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, Vol.39 Issue 6 (2004).
10) Kudriavtsev, A. and Kogge, P.: Generation of Permutations for SIMD Processors, *Proc. 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems* (2005).
11) Hennessy, J.L. and Patterson, D.A.（著），富田眞治，村上和彰，新實治男（訳）：コンピュータ・アーキテクチャ設計・実現・評価の定量的アプローチ，日経 BP 社 (1994).
12) Zivojnovic, V., Martinez, J., Schlger, C. and Meyr, H.: DSPstone: A DSP-Oriented Benchmarking Methodology, *Proc. International Conference on Signal Processing Applications and Technology*, Dallas (Oct. 1994).
13) Associated Compiler Experts bv. http://www.ace.nl
14) Barth, P.: A Davis-Putnam Based Enumeration Algorithm for Linear pseudo-Boolean Optimization, Technical Report, Max-Planck-Institut Fur Informatik (Jan. 1995).

(平成 19 年 7 月 6 日受付)
(平成 19 年 12 月 4 日採録)

田中 浩明（正会員）

平成 15 年大阪大学基礎工学部情報科学科卒業．平成 17 年大阪大学大学院情報科学研究科修士課程修了．現在，同大学院博士後期課程に在籍．組み込みプロセッサ向けのコンパイラのコード最適化の研究に従事．

武内 良典（正会員）

昭和 62 年東京工業大学工学部電気・電子工学科卒業．平成 4 年同大学院博士後期課程修了．博士（工学）．平成 8 年大阪大学大学院基礎工学研究科情報数理系専攻講師．現在，同大学大学院情報科学研究科准教授．ディジタル信号処理，VLSI 設計および VLSI CAD の研究に従事．IEEE，電子情報通信学会各会員．

坂主 圭史（正会員）

平成 9 年東京工業大学工学部電気・電子工学科卒業．平成 14 年同大学大学院博士後期課程修了．博士（工学）．平成 14 年大阪大学大学院情報科学研究科助手．現在大阪大学大学院情報科学研究科助教．VLSI レイアウト設計自動化，組み込みシステム開発自動化，設計最適化に関する研究に従事．

今井 正治（正会員）

1974 年名古屋大学工学部電気工学科卒業．1979 年同大学大学院博士後期課程修了（工学博士）．同年豊橋技術科学大学奉職．1994 年同教授．1996 年大阪大学大学院基礎工学研究科情報数理系専攻教授．その間，1984 年から 1985 年にかけて米国サウスカロライナ大学工学部電気計算機工学科客員助教授（文部省在外研究員）．これまで組合せ最適化アルゴリズム，ハードウェア/ソフトウェア協調設計等の研究に従事．1991 年より日本電子機械工業会および IEEE/DASC において EDA 標準化作業に従事．現在，情報処理学会設計自動化研究会主査．IEEE，ACM，電子情報通信学会，人工知能学会各会員．