情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 2　240–249 (Aug. 2008)

レジスタスロットを考慮した SIMD 向け細粒度自動並列化コンパイラ

三好健文[†1]　杉野暢彦[†2]

SIMD 演算は，高いデータ並列性を持つ計算処理に対して有効であるが，それらを有効に利用するプログラムが必要となる．そのため，アセンブリ言語や特殊な関数を呼び出すように変更することが要求される．しかし，明らかなデータ並列性のあるプログラムを変更することは容易であるが，内在しているデータ並列性を活用するためには，注意深くプログラムを解析するか，あるいは，実装するアルゴリズムそのものを再設計する必要がある．これは，大変困難で，また時間のかかる作業であるため，コンパイラによって自動化されることが望まれている．本論文では，SIMD 演算を手軽かつ効果的に活用するために，プログラム中に内在する並列性に着目した自動並列化手法を提案する．提案手法では，データを揃えるために必要となる Shuffle 命令を削減することで，少ないオーバヘッドで高い並列演算性能を引き出す．提案手法を Cell B.E. の SPU を対象とする自動並列化に適用し，トイプログラムに対しては期待どおりの並列度を，また DSPStone のプログラムに対しては，提案手法を適用しなかった場合と比べて 1.565 倍，gcc および XL-C でコンパイルした場合と比べ，最大 1.529 倍，1.715 倍の実行速度の向上を実現した．

Fine-grain Parallelizing Compiler for SIMD Processor with Consideration of Register Slot Allocation

Takefumi Miyoshi[†1] and Nobuhiko Sugino[†2]

This paper proposes the method to exploit parallelism and optimize memory allocation for SIMD processors. Recently SIMD processors are preferred for multimedia applications. Usually, programs which have much inherent data parallelism exploit SIMD processors, however it is necessary to write program by assembly languages and/or extended functions of high level languages. Since this is difficult and time-consuming work, the compiler which can exploit inherent parallelism in sequential program for SIMD is required. The proposed method derives sets of instructions which have inherent parallelisms from given programs and packs them as SIMD instructions streams to execute them in terms of parallel manners. In order to derive efficient sets, the method evaluates trade-offs between their effectiveness and overheads, quantitatively. Especially, it is important to consider overheads by shuffling instruction which locates data to applicable position in a register. And memory allocation and alignment is also considered in order to reduce the cost of loading the data. The proposed method is implemented as compiler for SPU of Cell B.E. And for several example programs, the generated codes are evaluated in terms of execution cycles and these results prove the effectiveness of the proposed method.

1. はじめに

SIMD プロセッサを有するプロセッサは，内部にデータ並列性を持つ計算処理に対して高い計算性能が得られる．一方で，多くの SIMD プロセッサでは構成を簡単にするために複雑なデータ処理機能や命令発行機能を持っていない．したがって，SIMD プロセッサを効果的に利用するためには，ソフトウェアがそれらを有効に利用するように記述されていなければならない．

多くの場合，SIMD 演算の性能を利用するためには，アセンブリ言語によるプログラミングを行うか，SWARC[1)] のような SIMD 演算を表現するために拡張された構文の利用，あるいは Cell Broadband Engine（Cell B.E.）[2)] における SPU プログラミングのように SIMD 演算にほぼ 1 対 1 に対応する固有の関数（Intrinsic 関数）を用意する方法[3)] などがある．また，SIMD 演算を用いるよう最適化された計算ライブラリがベンダから提供されている場合には，それらを適切に使用することで，比較的容易に SIMD プロセッサの高い計算性能を利用できる．

明らかなデータ並列性を有するプログラムや SIMD 演算部分を容易に切り出すことができるプログラムの記述を適切に変更することは比較的容易である．しかしながら，データ並列性が内在する逐次プログラムを効率良く実行するために記述を変更することは，注意深くプログラムを解析するか，あるいは，実装するアルゴリズムそのものを再設計する必要がある．これは，大変困難で，かつ時間のかかる作業であるため，コンパイラによって自動化されることが強く望まれている．

SIMD プロセッサを用いることで，マルチメディア処理のほか，一般の並列化可能な処理

†1 東京大学大学院創造情報学専攻
Department of Creative Informatics, The University of Tokyo
†2 東京工業大学大学院物理情報システム専攻
Department of Information Processing, Tokyo Institute of Technology

© 2008 Information Processing Society of Japan

に対して，低消費電力かつ少ないリソースで，高い演算性能を得ることができることが期待される．したがって，今後多くの場面で SIMD プロセッサが利用されることが予想され，コンパイラの需要はますます高まると考えられる．

従来より，SIMD プロセッサやマルチプロセッサアーキテクチャに対して多くの自動並列化手法が研究されてきた．特に SIMD プロセッサに対しては，自動ベクトル化によって大幅な計算速度の向上が期待されるため数多くの研究がある．その一方で，逐次プログラムを SIMD プロセッサ上で効率良く実現するための研究は，そう多くない．マルチプロセッサへの自動並列化手法は，プロセッサへの計算単位の割当ておよびそれらの計算順序をその全計算単位が終了するまでの時間を最小にする問題であり，数多くのアルゴリズムが提案されている．しかし，この問題は，NP 困難な難しい問題であるため，実際のプログラムに対して現実的な時間で解を得るためには発見的な手法が必要となる．

ところで，SIMD プロセッサ上でプログラムを並列実行する場合，演算ユニット間でのデータ転送をレジスタ内のデータ配置の変更として取り扱うことができる．これは，一般のマルチプロセッサシステムのデータ転送オーバヘッドと比較して小さい．そのため，データ転送オーバヘッドと並列性による速度向上のトレードオフを評価することで，十分に効果的な並列化コードを得ることができると考えられる．

本研究では，逐次プログラムを SIMD プロセッサ向けに自動並列化することを目標として，まずプログラム中の SIMD 演算可能性に着目した並列性を抽出し，できるだけその並列性を活用できるようにデータを分割した初期割当てを作成する．次に，その初期割当てに対する局所的なトレードオフに基づいたデータ転送のオーバヘッドを削減するための最適化手法を適用する．提案手法は，多項式時間の発見的手法による並列化であり，また，計算ユニットの個数や演算命令コストおよびデータ転送のコストをパラメータとして与えることができ，高い移植性を持つ．

本論文では，2 章で SIMD プロセッサについて述べ，3 章で提案手法である自動並列化手法について説明する．また，4 章では，簡単なトイプログラムの例に対して提案手法を適用することで適切に並列化されたプログラムが得られることを示し，ベンチマークとして DSPStone を用いて gcc および XL-C との比較によって提案手法の有効性を示す．

2. SIMD プロセッサ

本章では，SIMD プロセッサおよび，SIMD 演算におけるデータ転送のオーバヘッドについて述べる．

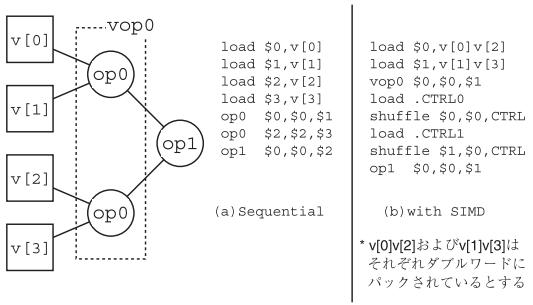

図 1　SIMD 演算によって高速化できない例
Fig. 1　An example of inefficient program for SIMD operation.

SIMD プロセッサは，一種のマルチプロセッサである．内部に持つ複数の演算器によって入力レジスタのオフセット位置によって区切られた各単位スロット（レジスタスロット）に対して同型の演算を並列に適用できる．この並列に実行される演算を SIMD 演算という．一般に SIMD プロセッサにおけるレジスタのデータ幅は大きい．たとえば，Cell B.E. の場合はレジスタのデータ幅が 128 bit であり，32 bit のデータに対しては SIMD 命令によって 4 並列での演算実行が可能である．しかし，SIMD 命令を用いて並列に演算を実行するためには，あらかじめ演算の対象となるそれぞれのデータを対応するレジスタスロットに適切に格納しておかなければならない．一般に，演算対象となるデータどうしがいつも適切なスロットにあるとは限らない．そのため，マルチプロセッサで演算に必要なデータの該当コアへの転送を必要とするのと同様に，SIMD プロセッサでも適切に演算を実行できるようにレジスタスロット間のデータ転送が必要となる．

SIMD プロセッサにおける演算器間のデータ転送はレジスタ上でのデータ割当て位置の変更によるものである．したがって，シフトやローテート命令などを用いたビット操作を用いてデータ転送を実現できるが，多くの SIMD プロセッサは，Pack 命令や Shuffle 命令などの専用の命令を持っている．たとえば，Cell B.E. の Shuffle 命令は，ソースレジスタの各ビットの値を制御レジスタの指示に従ってディスティネーションレジスタの任意の位置へ格納できる[4]．

データ転送は，SIMD 演算に対するオーバヘッドとなるため，できるだけその回数を削減することが望まれる．たとえば，**図 1** に示す演算木で与えられたプログラムを実行する場合を考える．破線で囲んだ 2 つの演算 `Op` が SIMD 演算可能な場合には，それらを独立

© 2008 Information Processing Society of Japan

に演算するか（図 1 (a)）SIMD 演算するか（図 1 (b)）を選択できる．ここで，図 1 (a) では，各変数は，128 bit 境界にアライメントされ，図 1 (b) では，v0v2 および v1v3 の順で，それぞれ 128 bit 境界の 64 bit プリファードスロットに配置されているとする．このように入力を工夫して SIMD 演算を用いた場合でさえも，Shuffle を行うための命令の挿入によるオーバヘッドによって図 1 (a) より図 1 (b) では，命令数が増加し，実行時間が余計に必要となる．

3. 自動並列化手法

本論文では，オーバヘッドとなるレジスタスロット移動の回数をできるだけ削減することを考慮した逐次プログラムの SIMD 向け自動並列化手法を提案する．SIMD プロセッサを含むマルチプロセッサに対する自動並列化は，一般に NP 困難な問題である．提案する手法では，プログラムを細粒度コード片に分割し，データの use/def 解析に基づいてレジスタスロット間のデータ転送をできるだけ少なくするようなコード片の初期配置および，局所的なデータ依存関係に基づく再配置手法によって，効果的な並列化を少ない計算量で実現する．

図 2 に提案する手法の処理ダイアグラムを示す．提案手法では簡単のために，内部に条件分岐を含まない基本ブロックだけを対象とする．また，レジスタスロット間の移動には，任意にビット位置を操作できる Shuffle 命令のみを利用することとする．

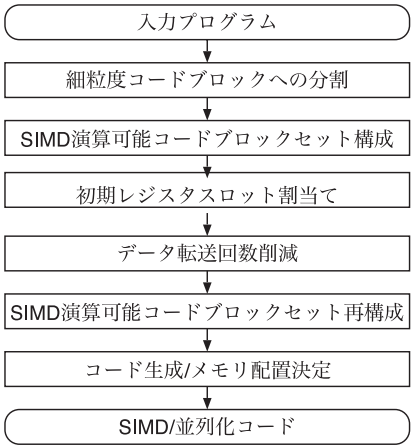

図 2　処理ダイアグラム
Fig. 2　A diagram of the proposed method.

提案する並列化手法では，与えられたプログラムを単項または二項演算程度の命令およびその代入からなる細粒度コードブロック（以下，コードブロック）に分割する．ここで，与えられたプログラムから得られた演算木が**図 3** (a) に示すようにバランスの悪い木であった場合，そのままコードブロックに分割すると高い並列性を抽出することが困難となる．そこで，並列性の抽出を容易にするために，結合則を満たす範囲で可換な演算を取り換えることで図 3 (b) に示すようなバランスの良い木へ変換し，コードブロックへ分割する．得られるコードブロック C は，自身の演算 $Op(C)$ および，データ依存による先行コードブロックおよび後続コードブロックの集合 $pred(C)$，$succ(C)$ から構成される．

以降，提案手法における処理を，**図 4** に示す小さな逐次プログラムに適用した結果を示しながら説明する．また，このプログラムをコードブロックに分割した結果を**図 5** に示す．ここで，グラフのノードである `C1` から `C7` は，与えられた図 4 のプログラムを分割することで得られたコードブロックであり，矢印はデータ依存関係を示している．

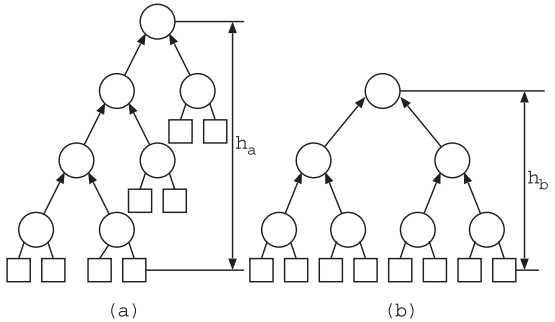

図 3　演算木の高さの比較
Fig. 3　Comparison of height of computation trees.

```
int test(int v0, int v1, int v2, int v3,
         int v4, int v5, int v6 int v7)
{
  return v0+v1+v2+v3+v4+v5+v6+v7;
}
```

図 4　例題：小さな逐次プログラム
Fig. 4　An example of a simple sequential program.

© 2008 Information Processing Society of Japan

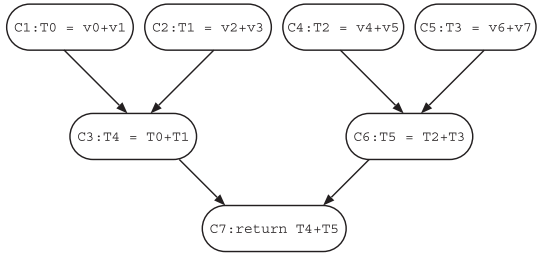

図 5　図 4 の細粒度コードブロック
Fig. 5　A fine-grain code block for Fig. 4.

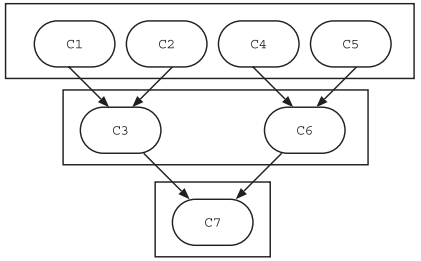

図 6　図 4 に対する SIMD 演算可能コードブロックセット
Fig. 6　A SIMD available code block set for Fig. 4.

3.1　SIMD 演算可能コードブロックセット

初期操作によって細粒度に分割して得られたコードブロックの集合を，SIMD 演算による演算が可能なコードブロックの集合に分割する．この集合を SIMD 演算可能コードブロックセットと呼ぶ．ここで，コードブロック C_0 と C_1 が同じ SIMD 演算可能コードブロックセットに属することができるのは，C_0，C_1 が並列に実行可能かつ同じ演算種を持つ場合である．しかし，実際には，C_0 と C_1 が並列に実行可能であることを決定するためには，命令スケジューリングとレジスタスロット割当てが必要となる．ここでは初期集合としての SIMD 演算可能コードブロックセットを簡単に得るため，近似的に C_0 と C_1 が

$$Op(C_0) = Op(C_1) \in \{Op_{SIMD}\} \tag{1}$$

$$\wedge\ {}^{\forall}C_{0_p} \in pred(C_0),\ {}^{\forall}C_{1_p} \in pred(C_1) \quad s.t.\ C_{0_p} \in \{C_d\} \wedge C_{1_p} \in \{C_d\}, \tag{2}$$

を満たす場合に同一の SIMD 演算可能コードブロックセットに属すると定義する．ただし，$\{Op_{SIMD}\}$ は SIMD 可能な演算種，$\{C_d\}$ は，与えられた演算木からトポロジカルソートによって得られた順序に従ってスケジューリングした結果，該当コードブロックを判定する時点ですでに実行可能である状態となっているコードブロックの集合である．C が属する SIMD 演算可能コードブロックセットを $set(C)$ で表す．

例題（図 4）から SIMD 演算可能なコードブロックセットを生成したときの結果を**図 6** に示す．図 6 中の四角の枠で囲まれているコードブロックどうしが，同一の SIMD 演算可能コードブロックセットに属している．

3.2　レジスタスロットの初期割当て

SIMD 演算を用いた効率の良い演算のためには，データ依存関係のあるコードブロックどうしのデータが同じレジスタスロットで計算されているとよい．一方で，SIMD 演算可能なコードブロックどうしのデータを同じレジスタスロット上に割り当てた場合，それらを並列に実行できない．そのため，初期割当てとしてデータ依存関係および属する SIMD 演算可能ブロックセットを考慮してレジスタスロット割当てを決定する．この段階では，無限個のレジスタスロットがあることを仮定する．それぞれのレジスタスロットは，ユニークな識別子で区別される．各コードブロックのレジスタスロットを決定するアルゴリズムは次のとおり．

(1)　レジスタスロット未割当てのコードブロック C を選択．なければ終了．
(2)　$pred(C)$ および $succ(C)$ の各コードブロックに割り当てられているレジスタスロットの識別子を列挙する．
(3)　列挙したレジスタスロットのうち，$set(C)$ 中のどのコードブロックセットにも割り当てられていないレジスタスロットがあれば，そのスロットの識別子をこのコードブロックに割り当てる．レジスタスロットがない場合，新しいレジスタスロット識別子を生成し，割り当てる．
(4)　$pred(C)$ に属するコードブロックの数が 1 のとき，そのコードブロックがレジスタスロット未割当てであれば，(3) で得たレジスタスロット識別子を割り当てる．同様に $succ(C)$ に属するコードブロックの数が 1 のとき，そのコードブロックがレジスタスロット未割当てであれば，(3) で得たレジスタスロット識別子を割り当てる．
(5)　ステップ (1) へ戻る．

コードブロック C が割り当てられたレジスタスロットを $reg(C)$ とする．

SIMD 演算可能ブロックセットが図 6 で与えらえれているとき，例題（図 4）のコードブロックは，このアルゴリズムの適用によって**図 7** 中に示すように割り当てられる．図 7 中，

© 2008 Information Processing Society of Japan

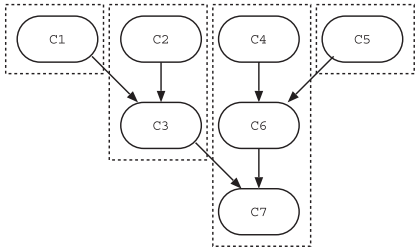

図 7　図 4 に対するレジスタスロットの初期割当て
Fig. 7　An initial register slot allocation for Fig. 4.

四角で囲まれているコードブロックが同一のレジスタスロットに割り当てられている．

3.3　データ転送命令の削減

初期割当てとして得られたレジスタスロット割当てでは，レジスタスロット間転送のオーバヘッドを削減する余地が残されている．これは，無用なレジスタスロットの分割が得られている可能性に起因する．そこで，必要となるデータ転送回数を削減するために，コードブロックのレジスタスロットへの割当てを変更する．本論文では，次に定義するデータ転送グラフを用いてコードブロック間の関係をモデル化し，そのデータ転送のオーバヘッドと割当ての変更による効果を定量的に評価する．データ転送グラフ上での操作は，プロセッサ間通信のオーバヘッド，プロセッサ数をパラメータとして与える．

本節では，まずデータ転送グラフについて述べ，次にデータ転送グラフを用いた再割当てのアルゴリズムについて述べる．

3.3.1　データ転送グラフ

データ転送グラフ G（**図 8**）は，プログラム中の演算単位とそのリソース割当てを取り扱うグラフであり，演算単位である計算ノード（computational node）N の集合，演算単位間の依存を示す辺（dependent edge）E の集合，およびプロセッサなどの計算リソース（resource）R の集合で定義される．

データ転送グラフ中のノードは，制約関係にあるノード N がどのリソースに割り当てられているかを，データ転送ベクトル（data transfer vector）$\mathbf{v}$ で保持する．たとえば，図 8 中のノード A は，リソース R_0，R_1，R_2 と制約関係にあるノードを，それぞれ，1 個，1 個，2 個持つ．そのため，データ転送ベクトル $\mathbf{v}$ は，$(1,1,2)^T$ と得られる．また，各リソースにおいて，他リソース間との通信にかかるオーバヘッドを重みベクトル（weight vector）$\mathbf{w}$ によって定義する．たとえば，R_1–R_0，R_1–R_2 間でのオーバヘッドを 1，R_1 内でのデータ転送オーバヘッドが 0 である場合，R_1 の重みベクトル $\mathbf{w}$ は，$(1,0,1)^T$ と得られる．データ転送ベクトル $\mathbf{v}$ および重みベクトル $\mathbf{w}$ のベクトル要素の並び順はリソースの順で一意に固定されているとする．該当するリソースのベクトル要素を選択する，あるいは選択しないためには，該当するベクトル要素が 1 で他の要素が 0 のベクトル $\mathbf{M_R}$ あるいは，逆に該当するベクトル要素が 0 で他の要素が 1 のベクトル $\overline{\mathbf{M_R}}$ との内積を求めればよい．

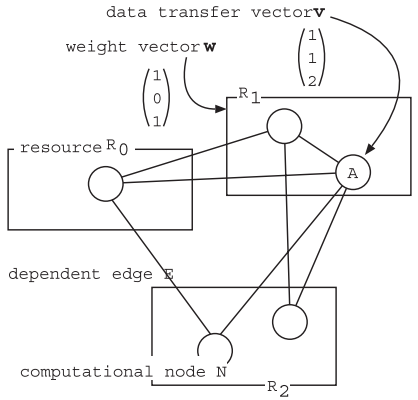

図 8　データ転送グラフ
Fig. 8　Data transfer graph.

ここで，ノード N のリソース R への割当てが適切かどうかを判定する局所的な評価値として転送ベクトル $\mathbf{v}$，$\mathbf{M_R}$ および $\overline{\mathbf{M_R}}$ を用いて，

$$I(N,R) = k{\mathbf{M_R}}^T\mathbf{v} - {\overline{\mathbf{M_R}}}^T\mathbf{v} \tag{3}$$

で与えられる親和度 $I(N,R)$ を定義する．ここで，k は N が接続相手として持つリソースの個数である．この親和度は，依存関係にあるノードのリソースに対する分布からのずれを意味し，この値が負の場合，ノード N のリソース R への割当てが不適切であると判定できる．また，転送ベクトル $\mathbf{v}$ を持つノード N をリソース R_0 から R_1 へ移動させた場合の評価値 P_R は，その演算コストの増加と，データ転送オーバヘッド増減量によって，

$$P_R(N,R_0,R_1) = {\mathbf{w_{R_0}}}^T\mathbf{v} - {\mathbf{w_{R_1}}}^T\mathbf{v} - S \tag{4}$$

で定義できる．ここで，$\mathbf{w_{R_0}}$ および $\mathbf{w_{R_1}}$ はリソース R_0，R_1 の重みベクトルであり，S は演算コストの増加すなわち移動にともなって並列実行ができなくなるコストを表す．

一般には，N がどの命令と SIMD 命令を構成するかはスケジューリングを行うまで決定

© 2008 Information Processing Society of Japan

できないため，この時点で S を導出することはできない．しかし，本論文では，この移動にともなって，同じ SIMD 演算可能コードブロックセットに属するコードブロックどうしが，同じレジスタスロットに割り当てられる場合，一律にその演算どうしの並列性が失われたこととして取り扱い，その演算時間をペナルティとして S に加える．

3.3.2　データ転送回数削減アルゴリズム

データ転送グラフを用いたレジスタスロット間の転送回数を削減する手順を以下に示す．ここで，コードブロックをノード N，データ依存関係を関係辺，そしてレジスタスロットをリソース R をとしてデータ依存グラフ上で取り扱う．データ転送回数削減アルゴリズムは次のとおり．

(1)　データ転送グラフをタスクグラフより作成する．
(2)　各計算ノードの親和度を求める．
(3)　負の親和度を持つ計算ノードがない場合，またはすべての負のノードが操作済みの場合に終了する．そうでない場合，最小の親和度を持つ計算ノード N を選択する．同じ親和度を持つ複数のノードが存在する場合には，データ依存の深さが最も浅いノードを選択する．
(4)　選択したノード N を，現在割り当てられているスロット R_0 から，評価値 $P_R(N, R_0, R_1)$ が最も大きくなるスロット R_1 へ移動させる．複数のスロットがある場合，最も早く実行を開始できるスロットへ移動させる．
(5)　選択したノード N を操作済みとしてマークする．
(6)　ステップ (2) へ戻る．

このアルゴリズムを，初期レジスタスロット割当てで得られた図 7 に適用した結果を**図 9** に示す．図 9 中の四角で囲まれたノードが，最適化アルゴリズムによって同じレジスタスロットへ割り当てられたノードである．図 7 の初期割当てと比較すると，C1，C4 および C5 のコードブロックが他のスロットへの移動の対象となり，データ転送回数が 3 回から 1 回に削減されていることが分かる．

3.4　SIMD 演算可能コードブロックセット再構築

SIMD 演算では，たとえデータ依存の制約において並列に実行可能なコードブロックであっても，同一のレジスタスロットに割り当てられたコードブロックどうしを並列に実行することはできない．そのため，コードブロックのレジスタスロット割当ての変更にともなって初期生成した同一の SIMD 演算可能コードブロックセットに属するコードブロックが SIMD 演算可能でなくなる可能性がある．そこで，C_0 と C_1 が同一の SIMD 演算可能

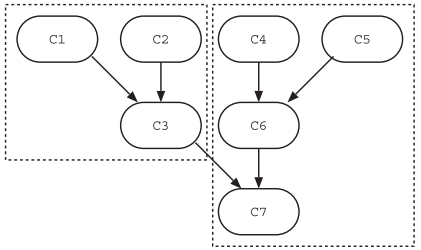

図 9　図 7 からアルゴリズムにより得たレジスタスロット割当て
Fig. 9　A derived register slot allocation from Fig. 7 by the proposed algorithm.

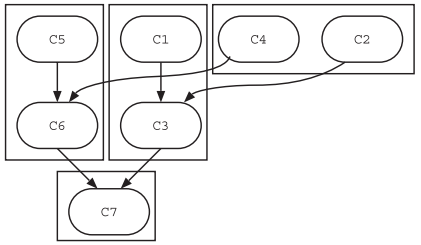

図 10　再構築後のコードブロックセット
Fig. 10　A code block set after restructuring.

コードブロックセットに属する条件として式 (1) に，

$$reg(C_0) \neq reg(C_1) \tag{5}$$

なる条件を加え，SIMD 演算可能コードブロックセットを再構築する．

レジスタスロット割当て最適化後の図 9 に示す状態に対して SIMD 演算可能コードブロックセットを再構築した結果を**図 10** に示す．ここで，図 10 の四角に囲まれたコードブロックどうしは，SIMD 演算可能なコードブロックである．図 6 と比較すると，レジスタスロット変更によって {C1，C2} および，{C4，C5} が並列実行の条件を満たさなくなったことが正しく反映され，コードブロックセットが分割されていることが分かる．

3.5　命令スケジューリング/メモリ配置最適化

得られた SIMD 演算可能コードブロックセットを基に，データ依存の深さが深いノードを優先するリストスケジューリングによって命令スケジューリングを行う．ここで，実際の SIMD 演算可能な幅を考慮しながら，SIMD 演算可能コードブロックセットの命令列を順に SIMD 化して出力される．さらに，各識別子に従ってレジスタスロットを決定する．

© 2008 Information Processing Society of Japan

また，ソースコード中から機械的に得られた順でメモリ上にデータを配置すると SIMD 演算を効率的に利用できない．これを回避するために，各変数および定数をあらかじめ適切なアライメントで配置する．

4. 評　　価

提案手法による実行コードの演算時間および処理時間を Cell B.E. SPU を対象として評価する．提案手法は，COINS[5] の LIR に対する操作として Java で実装し，抽出した並列性に基づいて LIR から Cell B.E. SPU 向けのアセンブリコードを生成する．生成されたコードの実行時間を，1 サイクルで 1 命令を実行すると仮定した命令レベルの簡易機能シミュレータ上での演算サイクル数を用いて評価する．提案手法では基本ブロック内部の自動並列化を取り扱うためこの範囲でパイプラインの乱れが発生せず，また Cell SPU がイン・オーダ実行のプロセッサであり，発行される命令の順序は変更されることがないことから，この仮定による実プロセッサとの動作サイクル数の乖離は十分小さい．ただし，gcc や XL-C では，パイプラインの乱れなどを考慮して NOP や分岐予測命令を生成している．これらは，この仮定では，不要な命令であり，これを考慮していない提案手法により生成されたコードと比較して，不利に作用する．そこで，これらの命令の実行サイクル数をカウントしないことで，用いるシミュレータ上での評価の公平性を保つ．

まず，提案する手法の効果を簡単なトイプログラムへの適用結果により確認し，次に，マルチメディア処理向けのベンチマークである DSPstone[6] に適用し，“gcc” および “XL-C” と生成コードの実行時間を比較する．評価には，MacOSX 10.5.1，プロセッサ Intel Core2Duo 2.2 GHz，メモリ 2 GB 667 MHz 上の Java2 1.5.0_13 および COINS-1.4.2.2 を用いる．

4.1 トイプログラムへの適用

提案手法によって，適切な SIMD 向けの並列化プログラムを得ることができることを**図 11** および **図 12** に示す簡単なトイプログラムへ適用して確認する．図 11 は与えられた変数の和を逐次に求めるプログラムであり，図 12 は，与えられた 12 個の変数から 4 個の和を求めるプログラムである．明らかに図 12 の最適な実行例の 1 つは，4 並列での実行である．

これらのプログラムの提案手法を適用した場合の処理本体部分の演算サイクル数とデータ転送回数およびそれらの合計である実行サイクル数を**表 1** に示す．ここで，(1) は提案手法を適用せず逐次で実行した場合，(2) は初期レジスタスロット割当てまでを適用しスケジューリングした場合，(3) は提案手法をすべて適用した場合である．並列度は逐次のプログラム（図 11）で 1.44 程度，並列プログラム（図 12）では期待どおり 4.0 の並列度が得られた．図 12 への処理の適用では，初期レジスタスロット割当て時には冗長な Shuffle 命令が挿入されたが，データ転送グラフを用いた最適化によって，この Shuffle 命令は削除され，期待どおり最適な出力が得られている．これは，提案する再割当て手法が有効であること示している．

```
int test0(int v0, int v1, int v2, int v3,
          ...
          int v68, int v69, int v70, int v71)
{
  return v0 + v1 + v2 + v3 + ... + v68 + v69 + v70 + v71;
}
```

図 11　シンプルな逐次プログラム
Fig. 11　A simple sequential program.

```
void test1(int a0, int b0, int c0, int d0,
           int a1, int b1, int c1, int d1,
           int a2, int b2, int c2, int d2,
           int a3, int b3, int c3, int d3)
{
  volatile int e0, e1, e2, e3;
  e0 = a0+b0+c0+d0;  e1 = a1+b1+c1+d1;
  e2 = a2+b2+c2+d2;  e3 = a3+b3+c3+d3;
}
```

図 12　シンプルな並列プログラム
Fig. 12　A simple parallel program.

表 1　図 11 および図 12 への適用結果
Table 1　Results of application to Fig. 11 and Fig. 12.

		(1)	(2)	(3)
逐次プログラム (図 11)	演算サイクル数	72	47	36
	Shuffle 回数	0	13	7
	合計実行サイクル数	72	73	50
並列プログラム (図 12)	演算サイクル数	12	3	3
	Shuffle 回数	0	1	0
	合計実行サイクル数	12	5	3

4.2 DSPstone プログラムによる評価

DSPstone は，マルチメディア処理でよく使用されるプログラムによるベンチマークセッ

© 2008 Information Processing Society of Japan

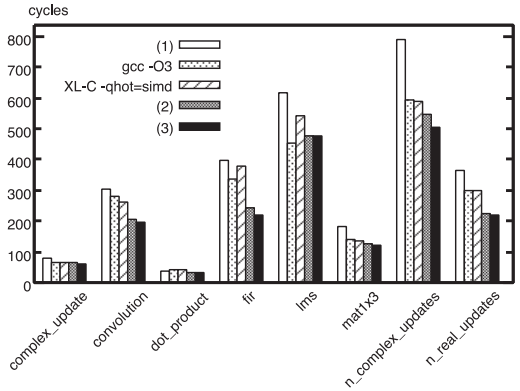

図 13　生成したコードの実行サイクル数の比較
Fig. 13　Comparison results of generated code execution cycles.

トである．このベンチマークのプログラムは，マルチメディア処理の特徴に起因する SIMD 演算に適した並列性を有しているが，明示的に SIMD 演算にあてはまるようには記述されていない．これらのベンチマークプログラムに本手法を適用し評価することで，逐次プログラムに内在する並列性を活用できるかどうかを評価できる．

DSPstone の固定小数点向けベンチマークに提案手法を適用した実行コードと “gcc -O3(gcc 4.1.1)” および，“XL-C -qhot=simd(Ver. 9.0) *1” でコンパイルして得られた実行コードのサイクル数の比較を**図 13** に示す．図 13 中，(1) は提案手法を適用せず逐次で実行した場合，(2) は初期レジスタスロット割当てまでを適用しスケジューリングした場合，(3) は提案手法をすべて適用した場合である．ここで，与えるプログラムは，ループをすべて展開し，また最適化手法の実装上の制約から，ポインタアクセスは等価な配列アクセスへ機械的に置換している．また実際の計算処理に相当する，初期値の代入処理を除いた `START_PROFILING` から `END_PROFILING` を評価の対象としている．

結果より提案した手法が与えられたプログラムから効率の良い実行コードを生成できていることが分かる．特に，提案手法を適用した場合では，n_complex_updates プログラムに対して，提案手法を適用しなかった場合と比べ 1.565 倍，fir プログラムに対しては，gcc および XL-C でコンパイルした場合と比べ，それぞれ 1.529 倍および 1.715 倍，実行速度が向上した．また，提案手法の適用に要した時間は，最短は complex_multiply への適用で 0.015 秒であり，最長は fir2dim.c への適用で 31.195 秒であった．これは全探索による場合と比べて非常に短時間で実現できている．

5. 関 連 研 究

SIMD 演算を活用する手法には，ベクトル演算向けに並列性を明示的に記述可能なプログラミング言語[7] を用いる手法や自動ベクトル化手法[8] など，従来より様々なレベルやアプローチがある．ここでは，逐次プログラムを SIMD プロセッサ向けに命令レベルで並列化する研究として関連の深い研究について述べる．

逐次プログラムを並列化する手法である Superword Level での並列性[9] を活用する自動並列化では，ベクトル化や命令レベルの並列化を統一的に取り扱い，効率の良い SIMD 実行グループの抽出とコード生成を実現する．与えられたプログラムから生成したデータフローの use/def 解析に基づき，レジスタ間の転送命令による負荷をなるべく小さくすることで，効率良い SIMD 演算を実現する．しかし，文献 9) の手法では，SIMD 演算可能な条件を満たす命令群候補が複数ある場合，そのコストによって選択を行っている．そのため，局所的な最適解におちいるという問題がある．

一方，文献 10) では，文献 11) のコード選択手法を拡張してデータフロー解析を基に SIMD 命令になる候補を抽出している．この手法では，機械語の選択において SIMD 可能命令どうしにおける命令選択の条件を制約式として与える整数線型計画問題として解いている．そのため，DLX を対象として DSPstone の FIR に対し 1.134 倍の速度向上を実現しているが，その処理時間に 5679.00 秒を要している．また，論文では SIMD 並列性が 2 の場合について述べてあるが，4 やそれ以上の場合には，制約式の個数が爆発的に増大し，現実的に解きえない可能性がある．提案手法における再割当て手法は多項式時間で実現することができるため，この手法と比較して，提案手法の再割当て処理を高速に解を求めることができるといえる．

SIMD プロセッサを効率良く利用するためには，データのメモリ上での配置も重要な問題であり，たとえば，連続するデータを効率良くロードするために配置を最適化することで高い計算性能を得ることができる[12]．複数の基本ブロックや全体の入出力を考慮する場合には，必要不可欠となる．

⋆1 配列内の連続的なエレメント上の特定のループ演算を SIMD 命令に置換する最適化オプションを有効にするオプション．

© 2008 Information Processing Society of Japan

6. ま と め

SIMD プログラムにおいて逐次プログラムを自動並列化するための発見的な手法を提案した．提案手法は，局所的な評価値に基づいてレジスタの再配置を行うため，実行速度が速い．また，提案するアルゴリズムでは演算ユニット数や命令とデータ転送のコストをパラメータとして与えることができ，様々なプロセッサへの応用が容易である．提案した手法を Cell B.E. を対象とした自動並列化に適用し，簡単なシミュレータによって評価した．トイプログラムに対する適用では，逐次プログラムから並列度を引き出すことができ，特に，明かな並列プログラムに対して期待どおりの適切な結果を得た．さらに，DSPstone のいくつかのプログラムへ適用し，効果と処理時間を評価した．

本手法は，局所的なオーバヘッドに着目した発見的手法である．したがって，レジスタスロットを移動した際の命令実行コストの増減に対して，最適解との差についての検討を深めることが今後の課題である．また，さらなる性能向上のために，文献 13) などの手法を用いてより多くの命令レベル並列性を与えられたプログラムから抽出することや，内部のパイプラインレベルを考慮して効率の良い命令スケジューリングを得ることなどが考えられる．提案するコストの評価手法は高い移植性を持つことも特徴であり，今後，たとえば COINS コンパイラインフラストラクチャを活用した SIMD 命令生成[14] を行い，様々な実ターゲットアーキテクチャ上での適用と評価を行うことも検討している．

さらに，与えられたプログラムを SIMD 演算を用いて効率良く実行できるように自動並列化するためには，大域的な SIMD 化への拡張を考えることも必要である．提案手法では，基本ブロック内でのレジスタスロット割当てに着目しているため基本ブロックをまたぐ際に冗長な Shuffle 命令が必要となることが考えられる．これを解決するためには，大域的な use/def や変数の独立性，変数の生死を考慮してレジスタスロット割当てを決定するように拡張しなければならない．

参 考 文 献

1) SWARC: SIMD Within a Register C. http://www.ece.purdue.edu/~hankd/SWAR/Scc.html
2) IBM Systems and Technology Group: Cell Broadband Engine Programming Handbook Version 1.1 (Apr. 2007).
3) CBEA JSRE Series, Cell Broadband Engine アーキテクチャ用 C/C++言語拡張 Version 2.3 (Dec. 2006).
4) CBEA JSRE Series, Synergistic Processing Unit 命令セット・アーキテクチャVersion 1.2 (Jan. 2007).
5) COINS コンパイラ・インフラストラクチャ協会：COINS 並列化コンパイラ向け共通インフラストラクチャ．http://www.coins-project.org/
6) Zivojnovic, V., Martinez, J., Schlager, C. and Meyr, H.: DSPstone: A DSP-Oriented Benchmarking Methodology, *Proc. ICSPAT'94*, Dallas (Oct. 1994).
7) Lawrie, D.H., Layman, T., Baer, D. and Randal, J.M.: Glypnir: A programming language for Illiac IV, *Comm. ACM*, Vol.18, No.3, pp.157–164 (1975).
8) Nuzman, D., Rosen, I. and Zaks, A.: Auto-Vectorization of Interleaved Data for SIMD, *SIGPLAN Not.*, Vol.41, No.6, pp.132–143 (2006).
9) Larsen, S. and Amarasinghe, S.: Exploiting Superword Level Parallelism with Multimedia Instruction Sets, *PLDI '00: Proc. ACM SIGPLAN 2000 conference on Programming language design and implementation*, New York, NY, USA, pp.145–156, ACM (2000).
10) Tanaka, H., Kobayashi, S., Takeuchi, Y., Sakanushi, K. and Imai, M.: A Code Selection Method for SIMD Processors with PACK Instructions, *Software and Compilers for Embedded Systems*, pp.66–80 (2003).
11) Leupers, R.: *Code Optimization Techniques for Embedded Processors*, Kluwer Academic Publishers (2000).
12) Ren, G., Wu, P. and Padua, D.: Optimizing data permutations for SIMD devices, *PLDI '6: Proc. 2006 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, pp.118–131, ACM (2006).
13) Schlansker, M. and Kathail, V.: Critical path reduction for scalar programs, *MICRO 28: Proc. 28th annual international symposium on Microarchitecture*, Los Alamitos, CA, USA, pp.57–69, IEEE Computer Society Press (1995).
14) Suzuki, M., Fujinami, N., Fukuoka, T., Watanabe, T. and Nakata, I.: SIMD Optimization in COINS Compiler Infrastructure, *IWIA '05: Proc. Innovative Architecture on Future Generation High-Performance Processors and Systems*, Washington, DC, USA, pp.131–140, IEEE Computer Society (2005).

(平成 20 年 1 月 29 日受付)
(平成 20 年 5 月 12 日採録)

© 2008 Information Processing Society of Japan

三好　健文（正会員）

昭和 56 年生．平成 15 年東京工業大学工学部電気電子工学科卒業，平成 17 年同大学院電子機能システム専攻修士課程修了．平成 19 年同大学院物理情報システム専攻博士課程修了．博士（工学）．同年東京大学情報理工学系研究科特任助教．現在に至る．自動並列化コンパイラ，HW/SW 協調設計に関する研究に従事．IEEE，電子情報通信学会各会員．

杉野　暢彦

昭和 39 年生．昭和 61 年東京工業大学工学部電気電子工学科卒業，平成元年同大学院物理情報工学修士課程修了．平成 4 年同大学院電子物理工学専攻博士課程修了．博士（工学）．現在，東京工業大学大学院総合理工学研究科物理情報システム専攻准教授．VLIW や DSP 向けのコード最適化，ディジタル信号処理アルゴリズムの実現技術の研究に従事．IEEE，電子情報通信学会各会員．

© 2008 Information Processing Society of Japan