制御並列アーキテクチャ向け自動並列化コンパイル手法

酒井淳嗣[†] 鳥居淳[†] 近藤真己[††]
市川成浩[††] 小俣仁美[††]
西直樹[†] 枝廣正人[†]

オンチップマルチプロセッサ MUSCAT 向けに，逐次プログラムを自動並列化する手法を提案する．本手法は，制御フローを解析し，後続基本ブロックを先行してマルチスレッド実行させるようなコード変換を行う．その際，スレッド間の依存関係に応じて適切なスペキュレーション方式を適用し，プログラムの動作を保証する．本手法をコードトランスレータとして実装し，実アプリケーションを用いて評価した．トレース情報によらない完全な自動変換により，単体のスーパースカラ実行に比べ，最高 1.7 倍の性能向上を達成した．また，fork 先スレッドの選択とデータ依存解析の厳密化により，更なる性能向上が可能との見通しを得た．

Automatic Parallelizing Method for Control-parallel Multi-threaded Architecture

JUNJI SAKAI,[†] SUNAO TORII,[†] MASAKI KONDO,[††]
NARUHIRO ICHIKAWA,[††] HITOMI OMATA,[††] NAOKI NISHI[†]
and MASATO EDAHIRO[†]

We propose an automatic parallelizing method for our on-chip multi-threaded architecture MUSCAT. This method translates a given serial program into a multi-threaded one, in which basic-blocks are scheduled to be executed in parallel with their previous blocks. In order to guarantee the meaning of the program, appropriate speculation methods, if needed, are applied according to dependency types between threads. We have implemented this method in a code-translator, and made its evaluation on some application programs. Our evaluation results show that our translator gains up to 1.7 times performance boost over the single superscalar execution without any run-time information nor manual code-improvement. Also, these results imply that more performance boost can be achieved with some improvements on our fork strategy and data-dependency analysis.

1. はじめに

マイクロプロセッサはこれまで，パイプライニング，スーパースカラ，out-of-order 実行等，もっぱらハードウェアを高度化することで性能を向上させてきた．しかしハードウェアの複雑化が動作周波数引き上げを困難にするほか，命令レベル並列性の限界も指摘され，ハードウェア増強のみによる性能向上は曲り角にさしかかっているといえる．

これに対し，ハードウェアとソフトウェアが密に協調して性能向上を図るアーキテクチャが提案されている．なかでも，1チップでプログラムのマルチスレッド実行を行うアーキテクチャ（本論文ではオンチップマルチスレッドアーキテクチャと呼ぶ）は，VLSI 技術の進歩により実現の見通しが立ち，活発に研究が行われている．1990 年代前半から研究されてきた Multiscalar[1] をはじめ，Superthreaded Architecture[2]，SPSM[3]，SKY[4] 等は，ハードウェアによるマルチスレッド支援機構と，それを活用するコンパイラの組合せで高い性能を得ることを目指している．我々も，PE 間でのレジスタ継承等に工夫を凝らしたオンチップマルチスレッドアーキテクチャMUSCAT を提案している[5]．

オンチップマルチスレッドアーキテクチャでは，自動的にプログラムをスレッド化するソフトウェア（以下マルチスレッドコンパイラと呼ぶ）が従来以上に重要な役割を担う．その理由は，スレッド間のデータ依存保証や投機的なスレッド実行は，ハードウェアの支援機構を利用しつつソフトウェアが適切に指示しなけ

† NEC C&C メディア研究所
C&C Media Laboratories, NEC Corporation
†† NEC 情報システムズ
NEC Informatec Systems, Ltd.

ればならないが，オンチップマルチスレッドアーキテクチャでのスレッド粒度は小さいことが多く，これらの指示をすべてユーザが適切に行うのは現実的ではないためである．

マルチスレッドコンパイラでのコンパイル方針として，コンパイル時情報のみでコード変換する方針と，トレース情報をも用いる方針とがある．トレース情報があれば，スレッド間のデータ依存の有無や実行パスの選択確率等の情報を得てより効率的なコード生成を行える可能性が高まる．他方，トレース採取のためにあらかじめプログラムを実行しておかなければならず，そのための適切な入力データの選定やトレース情報のコンパイラへの取り込み方法等，検討を要する項目も多い．

我々はMUSCATアーキテクチャ向けに自動並列化を行うマルチスレッドコンパイラの研究開発を進めているが，まず第一版として，基本的なコード変換手法をコードトランスレータとして実装した．トレース情報やユーザ指示を用いない，完全な静的自動並列化により，4PE構成で最高約1.7倍の性能向上を達成した．また，若干の手動チューニングを施すことでさらに性能を向上させうることを確認した．これらの結果から，将来，マルチスレッド処理を効率良く行うハードウェア/ソフトウェア協調型アーキテクチャが十分実現可能であるとの見通しを得た．

本論文では，MUSCAT向けコード変換を行うコードトランスレータの構造とその評価結果について述べる．まず2章でMUSCATアーキテクチャにおけるスレッド実行モデルについて説明した後，3章でコードトランスレータの仕組みを詳しく述べる．4章では実プログラムを用いた評価結果を分析し，最後に5章でまとめと今後の課題について触れる．

2. MUSCATアーキテクチャ

MUSCATは複数のPEを1チップ上に集積したアーキテクチャ[5]である．各PEは単方向リング状に結合され，その方向に沿って，あるPEが隣接PEを制御する．レジスタセットとPC（プログラムカウンタ）はPEごとに独立であり，スレッド生成時にレジスタの値が継承される他は，PE間でレジスタ値を直接通信する機構は持たない．他方メモリ空間はPE間で共通であり，キャッシュを共有する．

MUSCATは，プログラムの制御フロー中で将来実行するであろう部分を「繰り上げ」，現在実行しようとしている部分と並列に実行することで性能向上を図る．コンパイラは，ブロックA，B，Cを順に実行する逐次プログラム（図1(a)）を図1(b)のように変換する．実行時，MUSCATのPE0はブロックAをスレッドとして実行を始め，FORK命令を実行して隣接するPE1を起動する．FORK命令を実行した時点でのレジスタファイルの値がPE0からPE1にコピーされ，PE1はブロックBの実行を開始する．PE0がブロックAの最後にあるTERM命令に到達すると，PE0でのスレッド実行は終了する．このようなMUSCATのマルチスレッド実行方式を，我々は「制御並列」と呼んでいる．また制御並列実行において，新たなスレッドを生成した側を親スレッド，生成された側を子スレッドと呼ぶ．

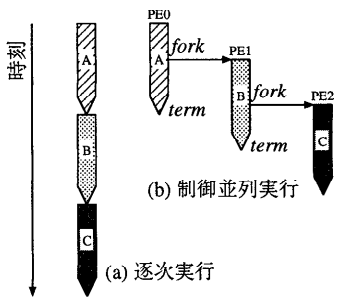


図1 制御並列実行
Fig. 1 Control-parallel execution.

MUSCATの制御並列実行におけるスレッド実行の単位は，数命令程度の基本ブロックから，数百命令程度の命令を含むような複数の基本ブロックの集合まで，様々な形態をとりうる．fork命令やterm命令といったスレッド制御操作を機械命令レベルで行えるため，細粒度スレッドでも効率的な並列実行が可能である．

2.1 制御スペキュレーション

図2(a)のような制御フローにおいて，ブロックA末尾にある分岐の分岐条件が確定する前にブロックBないしブロックCを開始することができる．このような，制御依存関係がある場合の投機的なスレッド実行を，**制御スペキュレーション**と呼ぶ．

MUSCATではSPFORK命令によって制御スペキュレーション状態のスレッドを生成する．コンパイラは，図2(b)に示すように，分岐条件確定前にスレッドを生成し，その後分岐条件が確定した段階で子スレッドの状態を確定（THFIX命令）あるいは破棄（THABORT命令）させるようなコードを生成する．

制御スペキュレーション状態でスレッド実行しているPEでは，ストア命令の結果はストアバッファ内に蓄積され，メモリへ反映されない．スレッド状態が確定されるとストアバッファ内の情報がメモリへライトバックされ，スレッドが破棄されるとストアバッファ

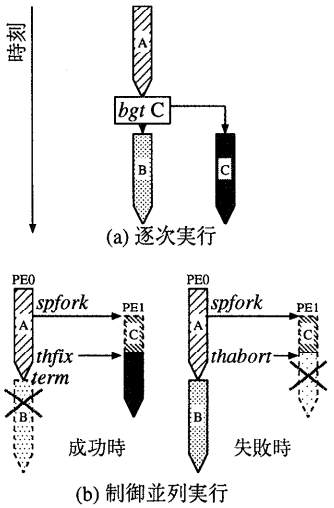


図 2 制御スペキュレーション実行
Fig. 2 Execution with control speculation.

の内容も破棄される．また，親スレッドが term 命令によって完了した場合は，子スレッドは自動的に確定状態となる．

2.2 データ依存関係の保証

スレッド間のデータ依存関係によって fork 前に必要な値が確定しない場合は，データ依存関係を保証するコード生成が必要になる．

データ依存を引き起こす要因は，親子スレッド間で共通のレジスタを使用する場合と，共通のメモリ領域を使用する場合の 2 種類に大別される．また依存の方向に関しては，親が書き込んだ値を子が読み込む正依存，子が書き込んだ値を親が読み込む逆依存，親子ともに同一ロケーションに書き込む出力依存の 3 種類がある☆．MUSCAT では，図 1 に示したように，逐次順では子スレッドよりも先に実行すべき部分を親スレッドに割り当てる方針をとる．つまり MUSCAT のスレッドモデルでは，親子スレッド間の逆依存および出力依存は，逐次実行では本来存在しなかった形式的なものである．

2.2.1 共通レジスタ参照によるデータ依存

親から子へレジスタが受け渡されるのは，fork 時のレジスター括コピーのみである．コンパイラは，正依存を引き起こすレジスタ，すなわち親から子へ受け渡すべきレジスタの値の計算が fork 前に完了するように fork 位置を決定する．

他方，fork 後は親子スレッド間のレジスタファイルは独立であるから，コンパイラはレジスタによる逆依存および出力依存について考慮する必要はない．

☆ ここでは一般的な並列化の場合と同様に，逐次プログラムでの参照順序を基に正依存，逆依存という用語を用いている．

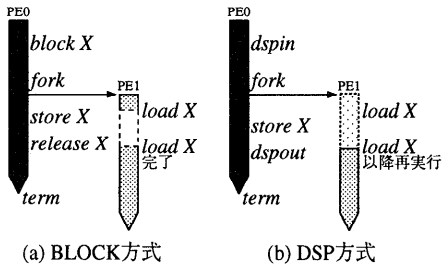


図 3 データ依存の対処方法
Fig. 3 Data dependency handling.

2.2.2 共通メモリ参照によるデータ依存

共有メモリ参照によるデータ依存に対して，MUSCAT では BLOCK 方式と DSP（データスペキュレーション）方式の 2 つの手法を用意している．

BLOCK 方式 BLOCK/RELEASE 命令は各々メモリの実効アドレスを指定し，当該アドレスにおいてメモリを介した正依存が存在することを宣言する命令である．BLOCK 命令で宣言されたアドレスに対して子スレッドがロード命令を発行すると，そのロード命令は親スレッドが RELEASE 命令を実行するまで完了しない．コンパイラは，

> 親スレッドは，依存アドレスに対して BLOCK 命令を発行した後に fork し，当該メモリへのストアが完了した後，RELEASE 命令を発行する

ようなコード生成を行う（**図 3** (a)）．このような実行方式を **BLOCK 方式**と呼ぶ．

BLOCK 方式では親スレッドがストアした値は子スレッドに伝えられるが，子スレッドがメモリにストアした値は親スレッドに影響しない．このような機能は後述するメモリ依存検出機構によって実現される．

DSP 方式 DSPIN/DSPOUT 命令は，ソフトウェアによる明示的なアドレス宣言なしにデータ依存問題を解決するための命令である．DSPIN 命令実行後に fork すると，子スレッドは**データスペキュレーション状態**☆☆と呼ぶ状態で実行される．この状態では，親子スレッド各々のメモリアクセスの衝突がメモリ依存検出機構（後述）で監視される．その後，親スレッドが DSPOUT 命令を実行したとき，もし子側が誤った

☆☆ データスペキュレーションという用語を「未演算の計算値を推定して投機実行すること」という意味で用いている研究もあるが，我々は「メモリアクセスを副作用なく投機的に実行すること」という意味でこの語を用いている．

ロード命令を実行していたことが分かれば，命令再実行機構（後述）によって子スレッドは衝突したロード命令から再実行される．このような実行方式を**DSP方式**と呼ぶ（図3(b)）．DSP方式においても，子スレッド側のストアは親スレッドに影響を与えない．

このように，大別してBLOCK方式とDSP方式という2つの方式がハードウェアで用意されており，両者には一長一短がある．コンパイラは基本的に，依存を引き起こすメモリアドレスをfork前に決定できるか否かによって，上記2つの手法を使い分ける．より詳細には，BLOCK宣言できるアドレスの数，BLOCKあるいはDSPINによる子スレッドの実行遅延や再実行コスト等，ハードウェア依存のパラメータも考慮して方式選択すべきである．

2.2.3 ハードウェア機構

前項で触れたメモリ依存検出機構と命令再実行機構の実現方法について簡単に説明する．

まずメモリ依存検出機構としては，メモリアドレスをキーとする連想メモリを持つ**ストアバッファ**を，データキャッシュとロードストアユニット間に設ける．ストアバッファはMultiscalarのARB[6]やSVC[7]と同様，各PEからのメモリアクセス要求に対し，スレッドの親子関係を加味した処理を行う．たとえば，親スレッドがストアする際には，子スレッドの連想メモリを検索し，子スレッドがそれ以前にロードを行っていたか否かを判定する．もしロードを行っていた場合，ストアバッファは命令実行シーケンサに指示を出し，当該ロード命令から子スレッドを再実行させる．

ストアバッファは子スレッドのストアの影響を親スレッド側に与えない働きも持つ．ストアバッファは子スレッドからのストア要求をその内部で処理し，親スレッドからのロード要求に対してはデータキャッシュの値を直接供給する．そのため，親スレッドは子スレッドのストア以前の値を得ることができる．

DSP方式時の命令再実行は，分岐予測失敗時の再実行と同様にリオーダバッファ（ROB）で行う．そのため，DSP方式で実行できる命令数は命令ウィンドウサイズに制限される．また実行できるストア命令数に関しては，ストアバッファエントリ数に制限される☆．

2.3 命令セット

MUSCATの命令セットは，ベースとなるRISC命令セットに制御並列関連の命令を追加したものである．制御並列拡張命令の一覧を**表1**に示す．

☆ 現在のところ，PEあたり最大4ストア命令程度を処理できる規模のストアバッファを検討している．

表1 命令セット 一覧
Table 1 Instruction set.

命令		動作
FORK	offset	確定 fork
FORKR	reg	確定 fork
SPFORK	offset	投機的 fork
THFIX		子スレッド実行確定
THABORT		子スレッド実行破棄
TERM		自スレッド終了，子スレッド確定
TERM*cond*	reg1, reg2	条件成立時のみ TERM 動作
BLOCK	offset(reg)	明示的データ依存アドレス指定
RELEASE	offset(reg)	BLOCK 指定解除
DSPIN		DSP 方式での実行指定
DSPOUT		DSP 方式での実行解除

3. コードトランスレータ

与えられた逐次プログラムを制御並列プログラムに変換する方法としては，(1) ソースレベルで並列化する方法，(2) 通常の逐次オブジェクトコードを並列変換する方法，および (3) 前記手法にプロファイルを採取して得たトレース情報を合わせて用いる方法，等が考えられる．並列化能力の観点からは (1) が最も有効と思われるが，コンパイラへの実装に手間取る可能性がある．我々はMUSCATアーキテクチャの有効性を短期間に評価するため，(2) の方式で，しかも前章で述べた手法を容易に実現できる「基本ブロック単位で変換するコードトランスレータ」方式を採用することにした．

この変換方法の基本アイデアは，図1におけるブロックとして基本ブロックを割り当て，基本ブロック末尾にある分岐命令をforkに変換することで，分岐先基本ブロックを現在の基本ブロックと並列実行させる，というものである．その際，トレース等の動的な情報は用いず，フロー解析等で得られた情報のみに基づいてコード変換を行う．

本コードトランスレータは，通常の逐次コンパイラが生成した実行プログラムのみを入力とし，トレース情報やユーザ指示なしで，fork等の制御並列命令を含んだプログラムを出力する．その処理概要を**図4**に示す．コードトランスレータは，まず，制御/データフローグラフを作成し，基本ブロック間の依存関係を調べ上げる．続いて制御フローグラフから基本ブロック支配関係を調べ，各分岐の種類（ループ繰返し/ループ脱出/その他）を判定する．そしてfork候補となる分岐を選定，forkへの変換を行った後，親子スレッド間の並列実行時間を増やすためにforkおよびその前後の命令のスケジューリングを行う．

以下，コード変換部分について，より詳細に述べる．

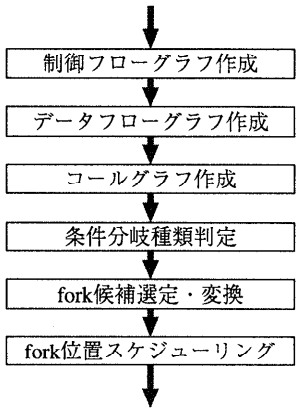


図 4 コードトランスレータの処理の流れ
Fig. 4 Process flow of the code translator.

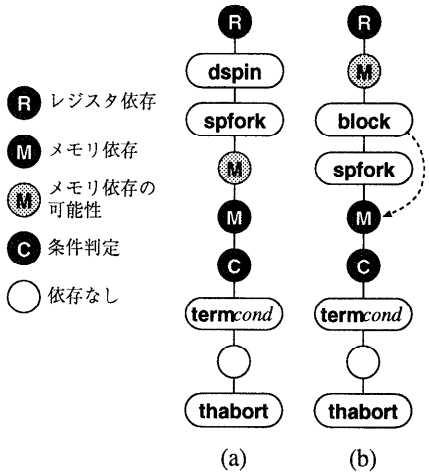


図 5 SPFORK 命令使用時の命令移動
Fig. 5 Instruction scheduling around spfork.

3.1 fork 箇所選定

各基本ブロック末尾にある分岐命令を fork 候補として調査する．条件分岐命令は SPFORK 命令，無条件分岐命令は FORK 命令への変換候補と見なす．

当該分岐命令を含む基本ブロックとその後続ブロックのデータフローを解析し，分岐命令を fork 命令に置き換えることで後続ブロックの実行開始が繰り上げられる命令数（これを **fork ブースト値**と呼ぶ）を見積もる．基本的には，fork ブースト値が **fork 閾値**と呼ぶ一定の値を超えたもののみ，上に述べた手法で実際に fork 命令に変換する．

fork 命令で生成した子スレッドが破棄されると性能向上率が低下するため，実行確率の高い基本ブロックを fork 先として選択することが重要である．条件分岐の中でも，ループの繰返しに相当する条件分岐は分岐確率に大きな偏りがある．本トランスレータはループ構造を検出し，ループの繰返し側を fork するようコード変換を行う．

3.2 データ依存への対処

メモリ参照によるスレッド間データ依存に対処する方法としては，BLOCK 方式と DSP 方式の 2 つの方式がある（2.2.2 項）．両者の選択は以下のように行う．

(1) fork 命令より下流に現れるストア命令の実効アドレス算出に使用されるレジスタをリストアップし，そのレジスタが fork 命令を含む当該基本ブロックの中で定義されている場合，DSP 方式を選択する．これは，BLOCK 命令でデータ依存の存在を宣言すべきメモリアドレスがストア直前まで決定されず，BLOCK 命令を fork 前に配置できない場合である．

(2) fork 命令より下流に出現するストア命令の実効アドレスの数が，ある一定数を超える場合，DSP 方式を選択する．

これは，BLOCK 命令でデータ依存箇所を宣言可能だが，必要な BLOCK 命令の数が多くオーバヘッドが無視できない場合である．

(3) 上記以外の場合は BLOCK 方式を選択する．

BLOCK 方式に比べ，DSP 方式は性能向上への寄与がプログラムの動的振舞いに影響されやすい．本トランスレータでは DSP 方式を強制的に抑止するモードを設け，DSP 方式の有効性も調べることにした．

3.3 fork 命令のスケジューリング

マルチスレッド実行による性能向上率を高めるには，できるだけ早い段階で子スレッドの fork を行うことが望まれる．そこで本コードトランスレータは，基本ブロック内で命令スケジューリングを行い，fork 命令を上流側へ移動させる．

まず，分岐を fork に変換しようとしている基本ブロック中で，レジスタ/メモリアクセスにより後続基本ブロックへデータ依存している命令に印をつける．メモリに関する依存については，依存が明確に存在するものと，依存が存在する可能性があるものとを区別する．依存のない命令は当該ブロックの下流方向に集め，その直前に条件判定命令群と term 命令を置く．その後の処理は fork 方式によって異なる．

- DSP 方式を適用する場合は，不確定なものも含めたメモリ依存命令を条件判定命令群の上流側に移動し，その直前に DSPIN 命令と fork 命令を移動する（図 5 (a)）．
- BLOCK 方式を適用する場合は，依存アドレスが確定しているメモリ依存命令を条件判定命令群の上流側に移動し，その直前に BLOCK 命令と fork 命令を

移動する（図5(b)）.
実際には，fork命令のスケジューリングとデータ依存への対処は密接に連係しながら進められる.

4. 評　　価

試作コードトランスレータの有効性を検証するため，実際のアプリケーションプログラム全体を本コードトランスレータで変換し，別途開発済みのアーキテクチャシミュレータ上で実行させてデータを採取した.

4.1 シミュレータ

評価時に用いたシミュレータは，MUSCATのパイプライン状態とキャッシュ状態を完全にシミュレートするもので，そのモデルはMIPS R10000相当のPEが4台集積しているオンチップマルチプロセッサである．単体PEの仕様を**表2**に示す.

4.2 評価プログラム

プログラム中にある程度の並列性が存在すると考えられるJPEG圧縮伸張プログラムcjpeg，djpegと，大きな粒度での並列性があまり期待できないcompress，eqntottの，合計4種類のプログラムを用いた．cjpeg，djpegはIndependent JPEG GroupのJPEGライブラリに含まれるもの，compressおよびeqntottはSPEC92ベンチマークに含まれるものを使用した.

評価対象プログラムはEWS4800のCコンパイラでコンパイルし，生成されたターゲットプログラムを今回作成したコードトランスレータに与えて制御並列プログラムに変換した.

4.3 評価結果および考察

4.3.1 性能向上率

本コードトランスレータで自動変換した結果を**表3**に示す．ここで

性能向上率　制御並列版と逐次版の実行サイクル数比，

スレッド生存率　生成された後，最後まで実行を完了できたスレッドの割合，

平均稼働PE数　同時に稼働していたPE数をプログラム全体にわたって平均した値，

である.

cjpegでは1.7倍程度の性能向上が見られる．これは総実行時間に占める割合の高い関数jpeg_fdct_islow()およびrgb_ycc_convert()内のループ構造が，データ依存保証用にそれぞれDSP方式，BLOCK方式を適用することで，ほぼ完全に制御並列化されているためである．PEの稼働率も比較的高い値を示している.

djpegにおいても性能は1.7倍程度向上している．cjpeg同様，実行頻度の高い関数ycc_rgb_convert()内のループが制御並列化されている点が効いている．cjpegよりPE稼働率が低いにもかかわらずcjpegと同レベルの性能向上が得られた要因の1つとして，破棄されるスレッドの割合がcjpegより少なかった点が考えられる.

compressでは性能向上率は1.2倍程度である．compressでは実行時間の半分以上が関数compress()で費やされる．この関数内には大きなwhileループがあるが，その内部の制御構造はかなり複雑である．そのためループ構造全体を自動で制御並列化できた部分はまったくなく，小さな分岐構造を制御並列化してわずかずつの性能向上を積み重ねているにすぎない．加えて，スレッド生存率がかなり低い．そのため，PE稼働率からみるとdjpegに匹敵するものの，全体としての性能向上率はかなり小さくなっている.

eqntottではさらに性能向上は小さく，1.1倍程度しかない．compress同様，制御並列化できるループ構造がないため，基本ブロック単位の小規模な並列化のみ行われている．表3の平均稼働PE数から考えると，その小規模な並列化を行っても，複数スレッドが並列に実行される部分はかなり少ないようである．さらに，スレッド生存率は1割強ほどしかなく，制御スペキュレーションで生成したスレッドのほとんどが無効になっている.

4.3.2 DSP方式の効果

DSP方式によるデータ依存保証は，ハードウェアとしての実装コストが比較的大きいほか，命令発行が命令ウィンドウサイズに制限される（2.2.2項参照）等，

表2　各PEの仕様
Table 2　Specification of each PE unit.

項目	パラメータ
パイプライン	IF,ID,Issue/Reg,EX,WB,Graduate Issue/Reg–WBをout-of-order実行
命令ウィンドウ	32命令/PE（整数16, L/S 16） 4命令同時デコード/終了
演算器	ALU×2, L/Sパイプ×1
ロードストア	3サイクルレイテンシ
分岐	4分岐まで仮実行， 履歴2048エントリ（PE間共有）
fork	1サイクルで実行
キャッシュ	命令/データ分離，各32KB 4way set associative（PE間共有）

表3　自動変換結果
Table 3　Results of automatic translation.

	性能向上率	スレッド生存率	平均稼働PE数
cjpeg	1.675	0.591	2.794
djpeg	1.679	0.710	2.380
compress	1.199	0.234	2.229
eqntott	1.095	0.143	1.255

表 4 DSP 方式を有効化した場合の改善率
Table 4 Improvement with DSP method.

	性能向上率	スレッド生存率	平均稼働 PE 数
cjpeg	+6.1%	+1.7%	+16.0%
djpeg	+1.3%	+0.6%	+7.2%
compress	+2.0%	+7.0%	−0.3%
eqntott	−0.7%	+1.4%	+2.5%

表 5 DSP 方式抑制による影響
Table 5 Effect of DSP method suppression.

	fork 消滅箇所（遅延平均）	fork 後退箇所（遅延平均）
cjpeg	6.0%（35.7 命令）	13.7%（7.4 命令）
djpeg	8.6%（20.3 命令）	12.1%（7.8 命令）
compress	17.9%（6.4 命令）	28.2%（2.8 命令）
eqntott	20.4%（6.9 命令）	13.9%（3.6 命令）

（括弧内は後続ブロックの開始時刻遅延の平均値）

現実にはトレードオフを考慮すべき点が多い．そこで，ソフトウェア側で DSP 方式を使う場合と使わない場合の比較を行った．本コードトランスレータは通常，3.2 節に示した手順で BLOCK 方式か DSP 方式を選択する．ここでは，DSP 方式の選択を強制的に抑止し，可能な限り BLOCK 方式を適用するモードでコードトランスレータを動作させ，通常のコードと比較した．測定結果を**表 4** に示す．

全体的な性能向上率でみると，cjpeg で 6%程度改善されている．これは，cjpeg の主たるループ構造が DSP 方式で効果的に制御並列実行されているためと考えられる．compress ではスレッド生存率が改善されたものの性能向上への寄与が少ない．これは本質的にスレッド間データ依存が多く，しばしば子スレッドの再実行が行われるためと考えられる．その他のプログラムでは性能的に大きな差異は見られなかった．

次に，プログラム中に fork 適用箇所がどれだけあるか，という観点から DSP 方式の効果を調べてみる．**表 5** は，DSP 方式適用を強制的に抑止して生成されたプログラムを通常の自動変換で生成されたものと比較し，DSP 方式抑止によって fork が消滅あるいは fork 挿入位置が後退した箇所の割合，およびそれらの fork 消滅/後退による後続ブロック開始時刻の遅れの単純平均値（命令ステップ数単位）を集計したものである．これらはアセンブラソースに対して調査した静的な数値である．compress や eqntott では DSP 方式によって fork 箇所が 2 割ほど増え，cjpeg や djpeg では DSP 方式によって fork 位置をかなり引き上げ可能である．fork 適用箇所拡大が性能向上に結び付かない要因としては，スレッド間データ依存のほかに，スレッド生存率の問題が考えられる．つまり制御の移行確率の高い部分を fork すべきであり，これは今後の重要な検討課題といえる．

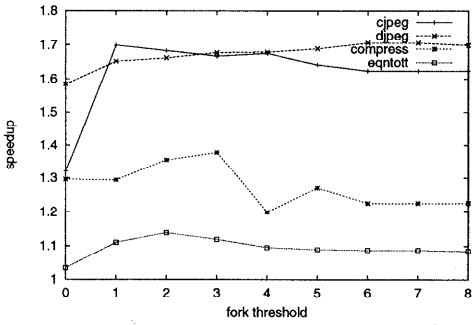


図 6 fork 閾値と性能向上率
Fig. 6 Fork threshold and performance gain.

4.3.3 fork 閾値の影響

図 6 に示すように，compress および eqntott の場合は fork 閾値 2 ないし 3 の場合に最も良い性能向上が得られている．これらのプログラムは制御構造が複雑なため，fork 閾値を上げると fork できる箇所が減ってしまうこと，また逆に fork 閾値を下げすぎると，無効となるスレッドが増えすぎることが要因と考えられる．compress および eqntott では fork ブースト値の平均は 7 ないし 8 命令であった．

それに対し，cjpeg，djpeg の場合は fork 閾値 0 の場合を除き，ほぼつねに最高性能に近い性能向上が得られている．その要因は，実行頻度の高いループ構造での制御並列化がうまく機能し，元の分岐命令よりもかなり上流の位置に fork 命令が挿入される場合が多いためだと考えられる．cjpeg，djpeg での fork ブースト値は 10〜13 命令程度である．

4.4 チューニングと今後の課題

この節では，本トランスレータでは未実装の機能に関し，人手で若干のチューニングを施して得た性能改善について述べる．

4.4.1 余分な DSPIN 命令の除去

十分並列性があると予想されるにもかかわらず 1.7 倍の性能向上にとどまっている cjpeg について考える．関数 jpeg_fdct_islow() にあるループはループアイテレーションを単位として DSP 方式で制御並列化されている．その実行状況をシミレーションログから分析してみると，データスペキュレーション状態で発行された命令がコミットできずにリオーダバッファ内に蓄積し，新たな命令発行ができない待ち時間が長く生じ

表 6 jpeg_fdct_islow() の手動改良
Table 6 Manual tuning in jpeg_fdct_islow().

	逐次版	自動変換版	DSPIN 削除版
性能向上率	1	1.894	3.674
IPC	1.783	3.424	6.600

表 7 compress の手動改良
Table 7 Manual tuning in compress.

	自動並列版	手動改良版
性能向上率	1.199	1.358
スレッド生存率	0.234	0.591

ていることが分かった．他方，このループ構造を厳密に解析すれば，ループアイテレーション間には実質的にデータ依存はなく，これはコンパイラレベルで判断可能と考えられる．

そこで手動で DSPIN 命令を外し，同関数だけを実行させた（**表 6**）．DSPIN を外すことでさらに 2 倍の性能向上が達成されている．また cjpeg 全体としても逐次比 2.0 倍の実行性能が得られた．このことから，親子スレッド間，特に性能向上に寄与しやすいループ構造での親子スレッド間のデータ依存をより厳密に解析する必要があることが分かる．

4.4.2 fork 先のチューニング

次に，自動変換での性能向上率が低い compress について分析する．前節で述べたように，compress ではスレッド生存率が低い．これは，条件分岐において，実際に選択される確率が低い方を fork 先として選択しているのが一因である．また，compress は制御構造が複雑で基本ブロック長が短く，条件分岐を fork に変換するだけでは十分な速度を得にくい．

そこで，compress の主たるループについて，別途採取しておいたプロファイル結果を基に「選択されない分岐先」への fork をやめ，複数の基本ブロックを単位とする fork を行うよう手動改良した．その結果，**表 7** に示すように，スレッド生存率が大きく上昇し，全体でさらに約 16%性能が向上した．このことから，制御構造が複雑なプログラムにおいては，条件分岐のどちらを fork するか，そして，複数の基本ブロックをまとめていかに効率良い fork 先を決定するかが重要なポイントとなることが明らかになった．

5. ま と め

逐次プログラムを制御並列アーキテクチャMUSCAT向けに変換する手法として，基本ブロックを順次引き上げて並列実行させる手法を提案した．この手法をコードトランスレータとして実装し，実プログラムに適用して評価した．その結果，トレース情報を用いない自動変換ベースで，R10000 相当のスーパースカラでの単体実行に比べ，最高約 1.7 倍の性能向上を得た．また本コードトランスレータによる変換結果を分析し，以下のような知見を得た．

- 比較的並列性の多いプログラムでは，データ依存解析を厳密化して余分な DSPIN 命令を減らすことで，性能を大きく引き上げることが可能である．
- 並列性の少ないプログラムではスレッド生存率が低く，性能向上率も小さい．実行確率の高いパスを fork するとともに，複数基本ブロックを単位とする fork に修正することで制御並列の効果を引き出すことができた．
- 動的なデータ依存チェックを行う DSP 方式は fork 適用箇所の拡大に役立つものの，それだけでは総合的な性能向上に結び付かない．
- fork 閾値の設定は，並列性の少ないプログラムの性能に影響する．今回のシミュレーションモデルではおおむね 2 ないし 3 程度が最適という結果になった．

制御並列アーキテクチャMUSCAT は，細粒度並列性の抽出処理の一部をハードウェアからソフトウェアに移し，ハードウェア/ソフトウェア協調によって，従来のハードウェアによる命令レベル並列の限界を超える性能向上を得ることを目指している．今回の評価および分析結果により，このようなハードウェア/ソフトウェア協調型アーキテクチャの性能を引き出すソフトウェア処理系が，十分実現可能であるとの認識が得られた．

ただし今回の評価では，一部のプログラムにおいては，自動変換ではまだ十分な性能向上が得られなかった．その原因として，粒度の小さな基本ブロックをスレッド化単位とし，条件分岐をそのまま fork 候補とするコード変換方針をとったことが考えられる．ソースレベルから変換を行うコンパイラとして実現する場合，複数の基本ブロックにまたがる効率的な fork 方式の決定が性能向上の鍵であると考えられる．そのためには，以下の項目について検討を進めていく必要がある．

より適切なスレッドの決定 命令実行コストや命令実行頻度を基に決定するのが基本だが，より詳細には，スレッド間データ依存による子スレッド実行停止ペナルティまで考慮したコスト評価が必要．

データ依存解析の厳密化 特にループアイテレーション間でのポインタや配列による依存の有無の見極め．

実行確率の高いパスの判定 静的解析だけでは分岐パターンを分類する程度しかできない．より高い性能

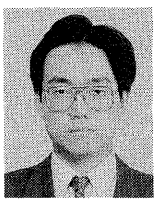
を引き出すには，一度実行プロファイルを採取し，それを fork 対象とするパスの判定に生かす方式も視野に入れて検討すべきである．

今後は自動変換コンパイラ作成を念頭に，上記項目，特に fork 箇所の適切な決定手法について，実プログラムの分析を行いつつ検討していく予定である．

参考文献

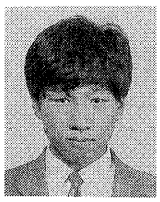
1) Sohi, G.S., Breach, S.E. and Vijaykumar, T.N.: Multiscalar Processor, *Proc. 22nd Annual Intl. Symposium on Computer Architecture*, pp.414 425 (1995).
2) Tsai, J.-Y. and Yew, P.-C.: The Superthreaded Architecutre: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation, *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp.35–46 (1996).
3) Dubey, P.K., O'Brien, K., O'Brien, K.M. and Barton, C.: Signle-Program Speculative Multithreading (SPSM) Architectire: Compiler-assisted Fine-Grained Multithreading, *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp.109–121 (1995).
4) 小林，岩田，安藤，島田：制御依存解析と複数命令流実行を導入した投機的実行機構の提案と予備的評価，情報処理学会アーキテクチャ研究会報告，I25-23, pp.133–138 (1997).
5) 鳥居，近藤，本村，池野，小長谷，西：オンチップ制御並列プロセッサ MUSCAT の提案，情報処理学会論文誌，Vol.39, No.6, pp.1622–1631 (1998).
6) Franklin, M. and Sohi, G.S.: ARB: A Hardware Machanism for Dynamic Reordering of Memory Refernces, *IEEE Trans. comput.*, Vol.45, No.5, pp.525–571 (1996).
7) Gopal, S., Vijaykumar, T.N., Smith, J.E. and Sohi, G.S.: Speculative Versioning Cache, *4th Intl. Symposium on High-Performance Computer Architecture* (1998).

(平成 10 年 8 月 31 日受付)
(平成 11 年 3 月 5 日採録)



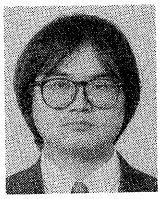
酒井 淳嗣（正会員）

1969 年生．1994 年京都大学大学院工学研究科情報工学専攻修士課程修了．同年 NEC 入社．同社 C&C メディア研究所にて自動並列化コンパイラの研究に従事．



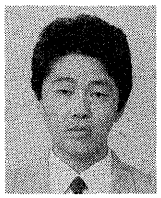
鳥居 淳（正会員）

1967 年生．1992 年慶應義塾大学大学院理工学研究科修士課程修了．同年 NEC 入社．同社 C&C メディア研究所にてマイクロプロセッサ，並列アーキテクチャの研究に従事．



近藤 真己（正会員）

1962 年生．1984 年山形大学理学部物理学科卒業．同年 NEC 技術情報システム開発入社．現在 NEC 情報システムズ オープン技術システム事業部にて自動並列化コンパイラの研究に従事．



市川 成浩

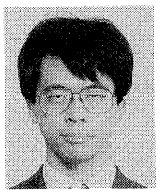
1964 年生．1990 年東京工芸大学工学部画像工学科卒業．同年 NEC 技術情報システム開発入社，自動並列化コンパイラの研究に従事．現在 NEC 情報システムズ オープン技術システム事業部所属．

小俣 仁美

1970 年生．1993 年電気通信大学電気通信学部情報工学科卒業．同年 NEC 技術情報システム開発入社．現在 NEC 情報システムズ オープン技術システム事業部にて自動並列化コンパイラの研究に従事．

西 直樹（正会員）

1959 年生．1984 年広島大学大学院システム工学専攻修了．同年 NEC 入社．スーパーコンピュータや並列コンピュータの研究/製品開発，またマイクロプロセッサ研究開発に従事．現在 NEC C&C メディア研究所研究マネージャ．



枝廣 正人（正会員）

1960 年生．1985 年東京大学大学院工学系研究科計数工学専門課程修了．同年 NEC 入社．VLSI レイアウト CAD や計算幾何学の研究開発，また並列アルゴリズム研究開発に従事．現在 NEC C&C メディア研究所主任研究員．