計算機アーキテクチャ 149－9
（２００２．８．２２）

Runtime Restructuring による複数コントロールフロー予測

玉造潤史[†] 平木敬[††]

Runtime Restructuring(実行時再構成方式) は、逐次バイナリプログラムをマルチプロセッサの各要素プロセッサに付加したハードウエアによって投機実行をする SPMD 命令流に変換し、そのプログラムを並列投機実行することによって高速化するアーキテクチャである。本論文では、Runtime Restructuring における投機実行モデル、データ投機、コントロールフロー投機の方式の詳細を示し、Runtime Restructuring と高精度の分岐予測方式を組み合わせることで、複数コントロールフローの投機的生成が実現されることを示す。この方式によって、SPEC CPU 95(fp/int) においての Runtime Restructuring の投機実行性能を示す。

Multiple control flow prediction with Runtime Restructuring

Junji Tamatsukuri [†] and Kei Hiraki[††]

Runtime Restructuringis an architecture accelerating sequential binary programs by dynamic speculative parallelization. It transforms sequectial programs to speculative SPMD program and carrys out in parallel by the hardware which added to element processors. This paper shows the details of the system of the speculative execution model, data speculation method, and control flow speculation in Runtime Restructuring. The combination of high precision branch prediction methods and Runtime Restructuringrealizes an effective speculative multiple control flow production. We show the performance of SPEC CPU95 (fp/int) in Runtime Restructuring.

1. はじめに

現在のマイクロプロセッサの速度向上には実行クロックの上昇、そして分岐予測精度の向上が大きく貢献している。局所的なデータ依存関係解析に基づく命令レベル並列性の活用はすでに上限に近いレベルで行われ、小さな基本ブロック内からも効果的に並列性を抽出している。しかしながら、命令レベル並列性には限界があり、プログラムから命令レベルより大きな並列性をどのように引き出すかがプロセッサの高速化の重要な問題である。特に、基本ブロックを越えた並列性抽出を実現するためには大きな命令スコープを持った並列命令発行ユニットを作らざるを得ず、それは設計上の困難を引き起こす。粗粒度の並列性において基本ブロックを跨る依存関係はコントロールフローに従うためフローの実行状態が確定するまで解決できず命令の並列実行を妨げる。つまり何らかの方法で命令実行時に単一のコントロールフローを越えた並列性を抽出し、実行できるフロー数あるいはサイズを拡大させない限り逐次プログラムの高速化には限界がある[1)]。

投機実行ではプロセッサの過去の実行履歴や現在の状態から先の実行状態を予測し、実行のフローを決め、先行実行を行う。予測が成功した場合に実行が高速化する技法である。単一のコントロールフローでの並列性の制限はプログラムカウンタに対するデータ依存関係が主要な原因であり、その予測を行うことで制限を越すことが可能である。また、投機実行ではプロセッサの実行状態を巻き戻すことが不可欠な機能となる。巻き戻し機能を用いることで並列実行時の他の値のデータ依存関係を同時に投機対象とすることができ、並列実行時に発生する同期関係の影響が減少する。分岐予測のような高精度の予測手法によって作られるコントロールフローに従って、実行する方向を決め、それらを投機的に実行することで計算資源を緩やかに同期した並列実行を可能とする[7)]。

コントロールフローを分割した動的なブロックレベルの投機実行を行うアーキテクチャは提案されており、逐次プログラムをターゲットとしたものでも、Multiscalar[8)]、Speculative Multithreaded Processors[12)] などがある。これらは Runtime Restructuring と同様、実行時に分割するブロックの依存性解析を行い、並列投機実行を行う。しかし、これらはマルチプロセッサが単一のプログラム流からコントロールフローは一つである。しかし、独立したプロセッサから構成され

† 東京大学 理学系研究科
University of Tokyo, Faculty of Science
†† 東京大学 情報理工学研究科
University of Tokyo, Faculty of Information Science and Technology

るマルチプロセッサでは高精度の分岐予測機構と正しい実行履歴を反映させることが出来れば独立の複数のコントロールフローを生成できる。つまり、並列の投機実行をプログラム制御については実行結果だけで同期させることでプロセッサ結合を減少させ、通信や実行の同期の少ない並列投機実行として実現できる。

本論文では、まず、逐次プログラムにコントロールフロー投機と投機データ並列性を用いたブロック並列性について述べ。ブロック並列性を活用するプロセッサアーキテクチャ Runtime Restructuring の複数コントロールフロー生成について示す。Runtime Restructuring では実行と並行して逐次プログラムを解析することにより投機ブロック実行を用いた投機 SPMD 並列プログラムに再構成する[13]。再構成した SPMD プログラムを高精度の分岐予測機構を用いて独立して各要素プロセッサで分散実行する。高精度分岐予測を用いた Runtime Restructuring アーキテクチャについてベンチマークを行い、性能を示す。

2. ブロックレベル投機実行

Runtime Restructuring は投機的にブロックレベル並列性を活用して実行時の並列化を実現する。ブロックレベル並列性とは、逐次のバイナリプログラムのコントロールフローを複数の基本ブロックごとに分割してつくられる実行ブロックを並列に実行することにより得られる並列性である。ブロック間には依存関係があるため、並列性の利用には次の条件を満たす必要がある。

(1) ブロック分割に利用できるコントロールフローがある
(2) 分割されたブロック間を跨る依存関係が正しく保たれる
(3) 他のブロックの実行のメモリアクセスに影響しない

この 3 つの条件を守ったまま、並列実行を行うことは困難であり、ブロックレベル並列性は投機的に利用せざるをえない。

投機的なコントロールフローの作成は分岐予測[2)3)] を用いることにより可能である。これらは分岐命令の現在までの実行のパスと分岐命令のアドレスに関連づけられたカウンタで予測する。分岐予測の結果に基づいてプログラムのフローを投機的に作成し、予測フローを分割することで投機的並列化ブロックを抽出することが可能である[6)]。

ブロック並列性におけるブロック間を跨るデータ依存関係は、図 1 に示すように大きく 2 種類に分類できる。すなわち、レジスタを介したものとメモリを介したものである。さらにこれらのデータ間依存関係にはコード生成時に、実際に使われ方によりさらに 3 種類に分類される。(a) レジスタ上のブロック間で頻繁に高速な通信を必要とする、(b) スタック上のローカルな状態を持ち、レジスタタよりも高速な通信を必要としない、(c) ヒープ上の実際に処理の対象となる大規模なデータである。プログラムはそれらを必要に応じて読み出し、実行する。(c) のようなデータが高速に同期してアクセスされる必要があるデータが並列のブロック間で共有される場合はロック動作が必要であり、(c) がない場合データ並列性があると言える。つまりブロックが使用するデータにデータ並列性がある場合にブロックレベルの並列性を抽出することができる[9)]。常に同期を必要とする依存関係 (Do Across) とならなければ (c) にあたるアクセスが起っても他のプロセッサの実行に間違った実行結果が副作用を起さないようにアーキテクチャが作られていれば良い。

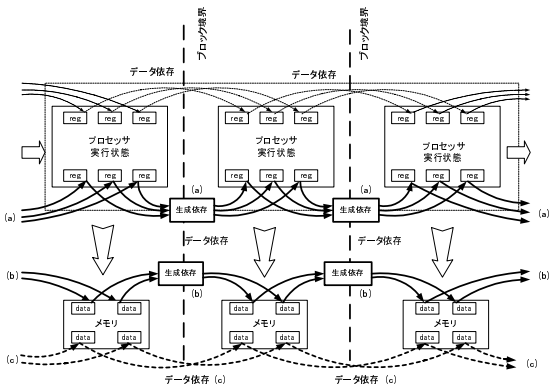

図 1 ブロック間依存関係

ブロック間のデータ依存関係を解消する方法には 3 つの方式がある。

(1) 生成したレジスタ (データ) を必要なブロックに転送する
(2) 値予測により生成する
(3) レジスタを生成する命令を抽出して繰り返し実行する

既存のブロックレベル投機実行を行うアーキテクチャでは (1) のレジスタ間接続[4)5)] を用いるか、(2) の値予測[12)] を用いるか、Runtime Restructuring のように (3) の生成によって実行状態を作り出し、投機実行の開始状態を作り出すかである。

レジスタ間接続による依存解決は、プロセッサ間での通信であり、データ依存関係に基づく同期関係となり、実行時にブロックは通信待ちを行うこととなる。値予測による実行状態の生成を行う場合では予測した値全ての予測結果が正しかったかどうかの判定が必要であり、実際に正しく生成した値はやはり次の実行に送らねばならない。これらはプロセッサ間通信必要とする。 Runtime Restructuring は、レジスタ値の比較を要する予測生成は行わずプログラムフローを予測実行し、実際にそのフローの予測が間違った場合に投機実行をやり直すという (3) の方式を用いる。プロセッサ間のレジスタ通信は存在せず、命令実行によって、投機対象をコントロールフローを分岐予測だけにとどめ、小さなハードウエアでの並列投機実行が実現される。

3. Runtime Restructuring

Runtime Restructuring(実行時再構成方式) は、マルチプロセッサの要素プロセッサが、実行時に動的に並列実行可能な SPMD スタイルのプログラムを生成しその投機並列ブロックを並列実行できるアーキテクチャである。並列動作のためのプログラムの解析は実行時に行うため、並列化もハードウエア実現できるようなサイズで実現される。

3.1 Runtime Restructuring の基本方式

Runtime Restructuring は、下記のような手順 (図 2) にしたがってプログラム再構成を行う。

(1) 逐次バイナリプログラムからターゲットとなる投機ブロックを発見する

(2) 投機ブロック内の生成依存性解析 (レジスタ依存解析) を行う

(3) 依存解析の結果から SPMD スタイルに再構成された投機的並列実行プログラムを生成する (投機並列化)

(4) 要素プロセッサは独立して並列投機実行する

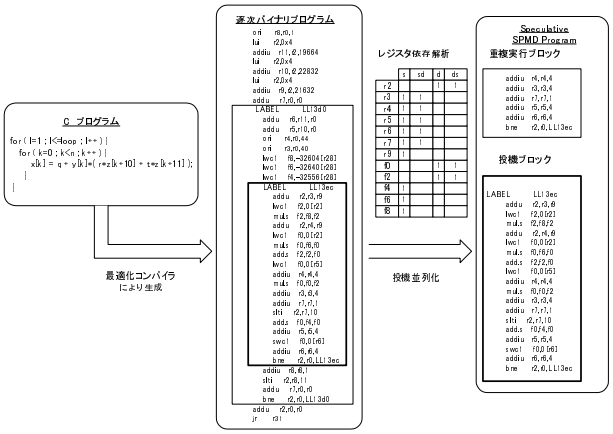

図 2 プログラム再構成アルゴリズム

投機実行の対象となるブロックの検出は分岐先ターゲットバッファに変更を加え、直前の分岐先エントリにプログラム構造として後続に来るプログラムフロー構造を保持できるようなエントリを付加したループターゲットバッファ (Loop TargetBuffer) で行う。検出したブロックは再度そのブロックを実行された時に検出に続いてブロック間依存解析が行われる。ブロック間依存解析は、命令デコードに並行し、生成依存関係をレジスタのアクセスの順序関係から検出する。レジスタに書き換えが行われた場合にそのレジスタは生成依存関係が存在し、そのレジスタを生成する命令を選択抽出、実行することでこれらのレジスタは投機的に生成される。ブロック間解析は 2 巡目以降、解析テーブル上のレジスタへのアクセスが書き込まれて変更し読み出すアクセスとなる依存をチェックする。解析テーブルが安定になったときに並列化が終了する。最終的に解析テーブルの内容は同じになり、レジスタ上の依存となる値を同期させることでき、ダイナミックな並列実行となる。解析によって抽出された生成命令を、新たな命令ブロックとし、命令バッファに格納する。このブロックを適当に繰り返し実行することでプロセッサ状態は投機的に生成される。この命令ブロックの繰り返し実行を「重複実行」と呼び、「重複実行」を用いることで、投機ブロック間の通信 / 同期が命令実行に変換され、重複実行終了時に実際に投機実行した結果のパスが正しかったことを確認することで投機の正否が判断できる。重複実行だけで投機実行でき、かつ、このプログラムはプロセッサ内で生成可能でき、並列性の粒度を大きくすることができる。

実行時解析によって作られるプログラムは要素プロセッサで同じものが作られるため投機 SPMD(SSPMD) プログラムと呼ぶ。 SSPMD プログラムは先に解析したテーブルに従って実行の内部状態を生成する命令ブロック (重複実行部分) とコントロールフローを作る命令 (分岐命令) とを含むブロック中における小さな命令ブロックを持つ。 SSPMD プログラムには同期や通信は明示されないが、逐次実行順序にしたがって行われるようにハードウエアを構成する。

3.2 Runtime Restructuring の動作モデル

Runtime Restructuring の SSPMD プログラムは各要素プロセッサごとに完全に独立したブロックとして実行され、他の並列実行されるブロックとの間では依存関係を解決するための通信を行わない。ブロック間の同期関係もメモリアクセスの順序、実行確定の順序が決められておりコントロールフローの投機実行となっている。メモリは投機的に書き換えを行うと他の並列ブロックの実行に影響してしまうため投機的にアクセスするメモリデータの依存関係を監視し、他の実行に副作用を起さないメモリアクセスが要求される。プロセッサの実行は投機開始時の状態を用いた予測からコントロールフローを生成し、投機実行の方向を決定する。投機的メモリアクセスに対するデータ依存関係の監視は、要素プロセッサ内に設置された投機アクセスバッファ (SAB) により実現される。実行結果を確定する際に投機メモリアクセスが失敗せず、予測したコントロールフローが正しいかったかどうかの判定をし、これらを通知することで正しい実行状態を保つ。

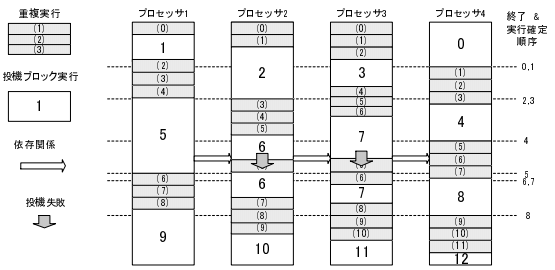

図 3 実行時再構成方式実行モデル

並列実行時の様子を図3に示す。

Runtime Restructuring の並列実行を行うプロセッサはそれぞれ独立した順序で実行される。要素プロセッサの投機実行に必要な資源が割り当てられる限り他のブロックの実行に無関係に実行を進める。投機実行開始時に、プロセッサは状態を保護するためにレジスタのデータをコピーする (リネーミングにより書き換え不能とする) 。投機実行を失敗したとき、投機実行開始状態に巻き戻るために必要である。実行中における直接的依存関係は「投機実行確定の順序」と「メモリアクセスの順序」である。どちらも実行の順序は逐次プログラムのコントロールフローにおける実行順序に厳密に従う。投機実行確定順序は投機実行管理機構 (Speculative Execution Manager : SEM) が、当該ブロックに先行する全ブロックの実行が終了することを判定し、当該ブロックの実行確定を行うことで守られる。他のプロセッサに当該ブロックの終了を通知するとともに実行パスを通知し、実行パスの確定を行う。投機的メモリアクセスバッファ (Speculative Access Buffer : SAB) は、データ依存性を壊すアクセスが行われた時に投機実行を失敗させる。ブロック内の投機メモリアクセスが失敗せず実行を確定をした直後のプロセッサの実行状態は実際に逐次プログラムが実行したときの当該ブロックの実行状態と同じである。重複実行と割り当てられたブロックの投機実行実行をおこない、投機ブロックから抜けた時に実行の終了を通知する。各プロセッサは終了が通知されたブロックの実行を終えた状態を重複実行を用いて作り出し、逐次実行への復帰する。

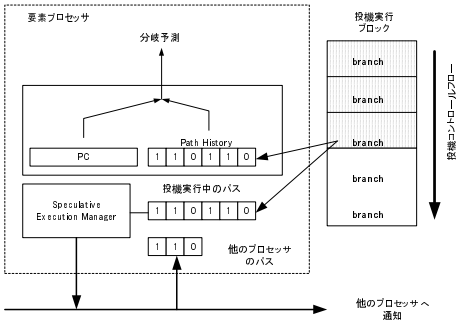

図 4 投機実行終了とコントロールフロー確定

今回の評価では、コントロールフローの予測に GSHARE の複合予測機構[2] を用いる。 GSHARE ではプログラムカウンタと実行パスから次の実行結果を生成する。その予測の成否結果を通知して正しい実行パスを各要素プロセッサ内に作ることで複数のコントロールフローを生成する。要素プロセッサの投機実行管理ユニット (SEM) は投機ブロックの実行終了時にブロックは実際に実行したパスを各要素プロセッサに通知し、実行の分岐予測に用いるパスとの比較を行う (図 4)。重複実行に含まれる分岐予測のパスに間違いがあれば実行はやり直す。並列投機実行を行いながら、正しい実行パスを分岐予測器が覚え、正しいコントロールフローを生成する。マルチプロセッサで複数の分岐予測器がある場合には有効な並列フローを生成できる。

4. Runtime Restructuring アーキテクチャ

Runtime Restructuring アーキテクチャは通常の対称マルチプロセッサ (Symmetric Multi Processor) に前述の投機ブロック検出、ブロック間依存解析、投機的メモリアクセスを実現する以下のような機能を付加することにより得られる。付加機能は、プログラム構造解析機構 (Loop Analyzer)、投機アクセスバッファ、投機実行管理機構であり、要素プロセッサに分散して付加される。

プログラム解析機構は構造の解析を行う Loop Table Buffer (LTB) とブロック間依存解析を行う部分、解析結果を格納する再構成バッファ (Runtime Restructuring Buffer:RRB) とを持つ。 RRB は再構成されたプログラムを格納する命令キャッシュである。 SAB は、ブロックレベル投機実行をのメモリアクセス順序を保証し、厳密に元の逐次プログラムの実行順序を守る。投機ブロック内における全てのメモリアクセスの履歴を残し、 Write に関してはバッファする。 Write に関しては Write Address を全要素プロセッサに通知し、 RAW 依存性を検出し、検出した場合には投機実行は失敗し、投機実行管理機構に通知する。

投機実行管理機構は、分岐予測を用いコントロールフローを予測生成し実行時の並列ブロックの実行の割り付けと実行の成否の検出、実行終了の通知および確定動作を行う。実行の終了、失敗の通知はバリア同期 (Elastic Barrier[14]) を用いて実現され、 Loop Analyzer からの投機実行の終了判定、 SAB からの投機実行のデータ依存検出による失敗通知を見て、投機実行の巻き戻しを行う。投機したコントロールフローの確認のために分岐予測機構で用いる GSHARE の実行パスを通知する。

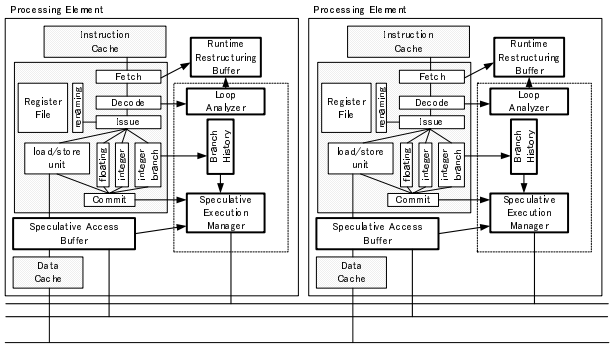

図 5 実行時再構成マルチプロセッサ OchaPro

これらの付加機能のついたテストベットアーキテクチャ OchaPro (On-CHip Architecture Processor) を図5に示す。

5. Runtime Restructuring の実行最適化

Runtime Restructuring ではハードウエアによって実行に解析されたレジスタ上の依存関係が重複実行によって解決され、並列実行される。プログラム解析に複数回の解析が必要なことと簡単な解析機構であるため解析が限定されることによって生じる性能ロスがある。プログラム解析の複数回解析については、前処理としてプログラム構造を解析し再構成を行うバイナリトランスレーションを行うことで解消することが出来る。解析機構の限界から来る性能ロスは、解析できないメモリ上のデータ依存関係が実行に大きく影響する場合である。レジスタ上にだけ依存関係が載る場合には、大きな並列性を引き出すことができる。しかし、レジスタ数には制限があるためブロックの実行時に用いられるデータメモリ上に割り当てられることがある。レジスタ上に置くべき緊密なデータ依存関係がスタック上に移ってしまう場合である。このような依存関係をもつメモリへ投機的にアクセスすると、同一のスタックは全てのブロックが読み書きをするためデータ並列性がないブロックとなってしまう。このようなスタック割り当ては Runtime Restructuring では問題となる。上にあるのと同様の生成依存関係を解析し、重複実行によって作られる必要がある依存関係である。つまり、並列性を引き出すためにこれらのレジスタと同様のアクセスを行うメモリを投機アクセスバッファで擬似的にレジスタと同じように実現する必要がある。

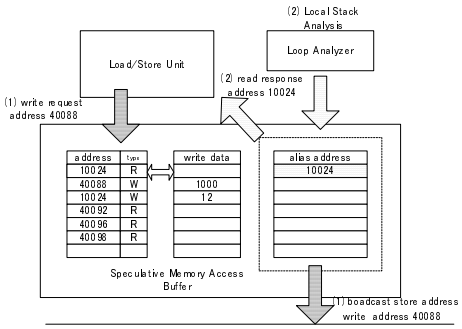

図 6 投機アクセスバッファでのメモリエリアシング

並列化解析時に、メモリロードによって生成されるレジスタをソースアクセスされたレジスタとしてテーブルを作成し、再構成された命令流に反映させる。並列実行時にはメモリに対するアクセスを投機アクセスバッファがブロックして値を返し、外部には通知しない。こうすることでそのメモリアクセスは投機アクセス失敗とならなくなり大きな性能向上が得られる。図 5 に投機アクセスバッファのメモリエリアシングの例を示す。通常のメモリアクセスは (1) のように外部に影響するが、(2) ループ解析の結果ローカルな値 (スタック値など) であることが分かれば、バッファがブロックして外部に出さない。そうすることでエリアスしたメモリでは投機実行は失敗しない。その結果効果的な投機実行となる。

6. ベンチマーク

Runtime Restructuring テストベット OchaPro シミュレータで実験を行った。要素プロセッサは Mips Architecture の 32 bit のプロセッサで整数ユニットを 2 個、浮動少数ユニットを 2 個持つスーパースカラプロセッサである。要素プロセッサには L1 命令キャッシュ / データキャッシュはそれぞれ 32KB,L2 キャッシュは 256KB あり、メモリとチップ内のクロック差は 4 倍である。並列投機実行時の system call は逐次的に行われ、全ての I/O アクセスはメモリアクセスにマップして行われる。投機機能については SAB は各要素プロセッサごとに 128Word 分あり、 RRB は 64 命令分持つ。逐次実行のプログラムは gcc-2.8.1 (最適化 -O2) でコンパイルし、バイナリトランスレーションによって生成したアセンブラをアセンブルして生成する。ベンチマークプログラムは、 SPEC 95 ベンチマークのうち INT から go,compress,li,gcc を FP から tomcatv, swim, mgrid を選んで実行した。入力は

6.1 実 行 結 果

実行結果は図 7,8である。グラフ中の opt はバイナリトランスレーションによる最適化を行った場合の結果である。データ並列性が十分にあり、コントロールフローも簡単な SPEC FP の tomcatv, swim では大きな速度向上率が得られる。一方、SPEC INT プログラムのうちコントロールフローが複雑で予測が困難な gcc では性能向上は 4% の性能向上になっている。

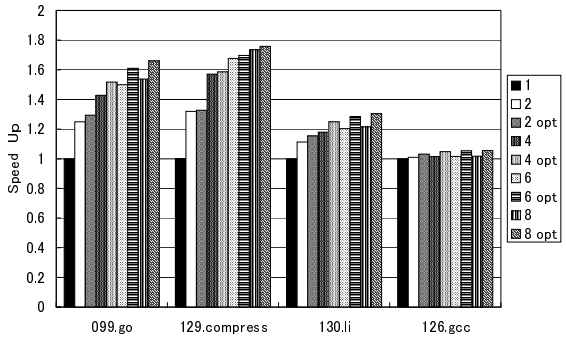

図 7 SPEC INT 95 結果

プログラムが使用するヒープ上のデータ並列性は十分に大きいことがこの結果から分かる。 SPEC FP の浮動小数レジスタは要素プロセッサの持つ数では足りないためメモリエリアシング最適化によってこの結果が得られている。 SAB におけるメモリアクセス最適化を行わない場合 (表 1) swim ではローカルな投機メモリアクセスの失敗が頻発し性能向上は得られない。レジスタの依存解析によって得られる性能向上が、レジスタ数の不足からスタック上へのあふれを起してしまうためである。特に、倍精度浮動小数点計算では、データ幅が大きくなるためレジスタの本数が減少し、レジスタ解析では収まらなくな

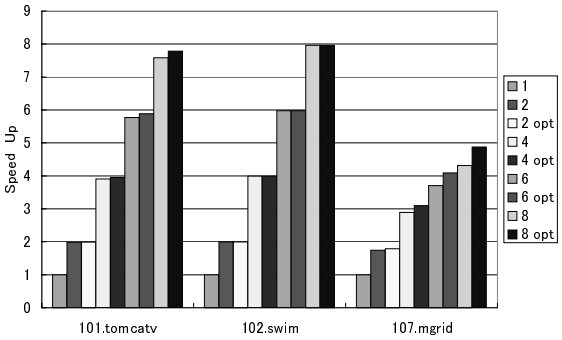

図 8 SPEC FP 95 結果

る。SPEC FP のアプリケーションのようにデータ並列性を多く含む場合でもメモリエリアシングは必要であることを示している。

	2 PE	4 PE	6 PE	8 PE
102.swim メモリ最適化あり	1.99	3.99	5.97	7.96
102.swim メモリ最適化なし	1.05	1.07	1.08	1.09

表 1 メモリエリアシング最適化による効果

また、バイナリトランスレーションによって得られた速度向上を表 2 に示す。この数値は最適化を行った場合の性能改善率を求めたものである。

アプリケーション	速度向上率 (%))
101.tomcatv	0.8%
102.swim	0.02%
107.mgrid	4.8%
099.go	3.8%
129.compress	0.5%
130.li	3.1%
126.gcc	2.2%

表 2 最適化による速度向上

特に、SPEC FP の tomcatv, swim では、実行時再構成したプログラムの実行が十分に行われるので、速度向上はあまり大きくない。一方 SPEC INT における li, gcc はコントロールフローが複雑で実行時の解析を行う際に大部分は並列実行前にフローが変ってしまう。そのため静的なフロー解析とコード生成は重要である。結果としてハードウエアだけでは速度向上が得られないがバイナリトランスレーションによって効果が得られる。

7. ま と め

本論文では、オンチップマルチプロセッサにおけるブロックレベル投機実行の実現方式として実行時再構成法を提案し、実行時再構成方式におけるブロック間依存を命令実行することによってコントロールフローとして投機する方式の詳細とその優位性について示した。また、チップ内での命令レベル並列性抽出の方式とその最適化法についても提案した。メモリ上に残る依存関係もメモリエリアシング最適化によって実行時再構成によって解決できる手法を示した。ベンチマークによる性能測定を行い実行時再構成方式の有効性とバイナリトランスレーションを用いた性能向上を示した。ブロックレベル並列性の抽出には、コントロールフローを投機して実行する手法が有効であるとともに重複実行のような並列実行を行うことでマルチプロセッサでのコントロールフローの投機並列化が行えることを示した。

謝辞

本研究の遂行にあたり多大なるアドバイスを頂いた 国立情報学研究所 松本 尚博士に深く感謝します。

参 考 文 献

1) M.S.Lam , R.P.Wilson, "Limits of Control Flow on Parallelism." , In proceedings of ISCA-19 , pp. 46 – 57, May (1992).
2) S.McFarling, "Combining branch predictors." , Technical Report TN-36 Digital Western Research Laboratory, June (1993).
3) J.Stark, M.Evers, Y.N.Patt, "Variable Length Path Branch Prediction" , In proceedings of ASPLOS-IIX, pp.170 – 179, October (1998).
4) T.N.Vijaykumar, S.E.Breach, G.S.Sohi, "Register Communication Strategies for the Multiscalar Architecture" , Technical Report #1333, University of Wisconsin-Madison, February (1997).
5) V.Krishnan, J.Torrellas, "The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors." , In proceedings of PACT'99 , pp.24 – 33.
6) J.Oplinger, D.Heine, M.S.Lam, "In Search of Speculative Thread-Level Parallelism." , In proceedings of PACT ' 99.
7) "Path-Based Next Trace Prediction.", Q.Jacobson, E.Rotenberg , J.E.Smith, In proceedings of MICRO-30 , pp.14 – 25, Dec (1997).
8) G.S.Sohi, S.Breach, T.N.Vijaykumar, "Multiscalar Processors." , In proceedings of ISCA-22, pp.414 – 425, (1995).
9) Y.Sazeides, S.Vassiliadis, J.E.Smith, "The Performance Potential of Data Dependence Speculation & Collapsing" ,In proceedings of MICRO-29, pp.238 – 248, Dec (1996).
10) M. Gupta and R. Nim, "Techniques for Speculative Run-Time Parallelization of Loops.", In proceedings of Supercomputing'98.
11) V.Krishnan, J.Torrellas, "Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor," , In proceedings of the 1998 ACM International Conference on Supercomputing, pp. 85 – 92.
12) P.Marcuello, A.Gonzalez, J.Tubella, "Speculative Multithreaded Processors," , In proceedings of the 1998 ACM International Conference on Supercomputing, pp. 77 – 84.
13) K.Hiraki, J.Tamatsukuri, T.Matsumoto., " Speculative execution model with duplication" , In proceedings of The 12th International Conference on Supercomputing'98, pp.321 – 328.
14) T.Matsumoto, T.Tanaka, T.Moriyama, S.Uzuhara, "MISC: a Mechanism for Integrated Synchronization and Communication using Snoop Cache" , in proceedings of 1991 ICPP,I-161.