計算機アーキテクチャ 134－18
（1999．8．3）

複数パス投機実行モデルの有効性の検証

大津金光† 吉永努† 馬場敬信†

近年の汎用マイクロプロセッサはクロック速度の向上と命令レベル並列性の活用により性能向上を果たしてきた。クロック速度は物理的限界、命令レベルでの並列度は制御依存による限界が存在するため、プロセッサのさらなる性能向上にはスレッドレベルでの並列性の活用が必須となる。

これまでに我々は投機的マルチスレッド実行を複数パスで行なうことによる従来の逐次プログラムの性能向上を目指した実行モデルを提案してきた。本稿では実際のプログラムに対して本提案モデルを適用し、シミュレーションにより性能評価を行なった結果を示す。

Effectiveness of Speculative Multi-threading with Multi-path Execution

KANEMITSU OOTSU,† TSUTOMU YOSHINAGA † and TAKANOBU BABA†

Recent microprocessors' performance has been highly improved by their high-speed clock frequency and by exploiting instruction-level parallelism. Physical limitation of clock speed and semantical limitation of control dependencies impede the improvement of the performance. To overcome this difficulty, it is indispensable to make use of thread-level parallelism.

On this background, we have proposed a speculative multithreading model with selective multi path execution and its implementation that aims at speed-up of usual sequentical program codes. This paper shows the example applications for the proposed model and evaluates the effectiveness of the model by simulation.

1. はじめに

近年の汎用マイクロプロセッサは半導体技術の進歩によるクロック速度の向上と命令レベル並列性 (**ILP: Instruction Level Parallelism**) の活用により格段に性能が向上してきた。クロック速度の向上には物理的限界が存在するため今後は並列性の活用が必要不可欠となるが、ILP の並列度は投機実行を導入しても高々5～10 程度[1] と言われており、マイクロプロセッサの性能向上にはさらに上位の並列性を活用が必須となる。ILP の並列度が小さいのは同時に一つの命令流 (シングルスレッド) のみを実行の対象としているからであり、複数命令流 (マルチスレッド) で実行を行なえば格段に並列度を上げることが可能となる[2]。これを背景として最近は制御フローに沿って投機的にマルチスレッド実行を行ない、並列度を向上させる研究が行なわれてきている。

プログラムの制御フローに沿った投機的マルチスレッド実行モデルは分岐点の片方のみを投機実行の対象とする単一パスモデル[3)~5)] と両方を実行の対象とする複数パスモデルの二つに分類できる。単一パスモデルの問題点は、投機実行を深く行なえば行なうほど実際にその実行結果が使われる可能性が小さくなっていくことである。最先行の投機実行の結果が実際に使われることが皆無という状況は十分にありえる。さらに単一パスモデルには分岐予測に失敗した場合のペナルティコストの問題がある。これに対して単一パスに沿って実行した結果が実際に使われる可能性が小さい投機実行を行なうよりは、その分の計算資源を予測とは異なる方向の実行に使用する複数パスモデルは資源の有効利用や分岐予測失敗時のペナルティコストの削減という観点から有用であると考えられる。

EE (Eager Execution) モデルは複数パスモデルの一種であり、プログラム上の分岐点における全てのパスを全て実行するモデルである。EE モデルは原理上分岐予測ミスによる性能劣化は存在しないが、その実現には指数的規模のハードウェア量を必要となる。

DEE (Disjoint Eager Execution)[6] モデルは実行確率の大きいところを優先的に実行するモデルである。このモデルではプログラムの制御構造を木構造として表現した上で、その根からの累積した分岐確率が大きいところが優先して実行される。通常、累積した分岐確率の計算は各分岐点に対応した乗算器などの膨大なハードウェア資源を必要とするため、文献 6) ではプログラム全体で分岐確率は一様であるという仮定を置いて計算を単純

† 宇都宮大学工学部情報工学科
Department of Information Science, Faculty of Engineering, Utsunomiya University

化している。

しかしながら、DEEモデルのこの単純化により局所的な分岐確率の偏りには対応できないという問題を生じる。一般に、プログラム中の条件分岐においてその分岐確率が全ての個所で一定ということはありえない。従って、一様な分岐確率という仮定を置いて投機実行を制御した場合には実行効率が低下する可能性がある。投機実行モデルにおいて投機実行に消費する計算資源は可能な限り少ないことが望ましく、無駄な実行となる可能性が高い実行に計算資源を使うことは避けるべきである。

そこで我々は各分岐点におけるスレッド制御を細かく実現することで無駄な投機実行を減らし計算資源の利用効率を高めることが可能な新たな複数パスモデルを提案してきた[7),8)]。本稿では実際のプログラムに本提案モデルを適用した場合にどれくらいの性能向上が達成できるかをシミュレーションにより評価した結果を示す。

2. 実行モデル

本稿で実際のプログラムに適用し評価を行なった複数パスによる投機的なマルチスレッド実行モデルについて説明する。図1の制御フロー木において各ノードは基本ブロックを表わし、各辺は2方向の分岐先を表わしている。さらに左辺側の基本ブロックが右辺側よりも実行確率が大きいものとする。

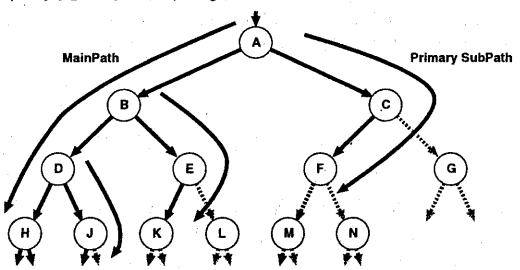

図1 実行モデル

ここで図の制御フロー木上のパスにおいて各分岐において全て分岐確率が高い方向を採るものをメインパスと呼び、途中の分岐のいずれか1回のみを実行確率の低い方向を採るパスを**1次サブパス**と呼ぶ。同様に途中の分岐の n 回を実行確率の低い方向を辿るパスを n **次サブパス**と呼ぶことにする。

図において実線の辺のパス上の基本ブロックコードを投機実行の対象とする。これらはメインパス上であるか、1次サブパス上であるかのどちらかである。

メインパス上の基本ブロックコードの全てをマルチスレッドで投機実行を行ない、1次サブパス上のコードはその先頭の一部のみをマルチスレッドでの投機実行を行なう。

この制御は、各スレッドから子スレッドを生成する際にメイン側サブ側のそれぞれに対して何代先の子スレッドまで投機実行を許可するかの情報を指定することで行なう。生成要求を出されたスレッドは先ずメイン側の子スレッドが無くなるまでプロセッサへ割り付けられた後、サブ側の子スレッドを割り付けられる。図においてノードAに対応するスレッドがメイン側に**L**個、サブ側に**R**個を指定した場合、メイン側の**L**個の子スレッドが全てプロセッサに割り付けられて実行を開始するまでサブ側のスレッドが実行されることはない。例えば**L**=3,**R**=1とした場合、制御木上のAのスレッドのメイン側B,D,Hの3個のスレッドとサブ側Cの1個のスレッドが投機実行を行なう。

L,**R**の値は分岐点での分岐確率に依存する。圧倒的にメイン側への分岐が多い場合は**L**を大きく**R**は小さくするのが妥当な戦略である。逆に、メイン側とサブ側への分岐の差が小さければ**L**,**R**はほぼ同じ値とする。

分岐先が確定し、メイン側の投機スレッドが実行されるべきであることが確定した場合、サブ側のスレッドの実行をその子孫を含めて全て無効化する。逆にメイン側の投機スレッドが実行されるべきでないことが確定した場合は間違っていたスレッド以降の全てのスレッド実行を無効化する。この場合、サブ側のスレッドの実行が正しいのでこちらを制御木上の新たなrootとして実行を継続する。ここでメイン側のスレッドの生成消滅にかかるコストによる性能低下はサブ側の投機実行により補償される。

新たにrootとなったスレッドは今度は自分が生成したメイン側およびサブ側の子スレッドに対する**L**,**R**の値によって以上説明したものと同様の制御を行なう。

ここまで制御フロー木を二分木とし、各ノードは基本ブロックとしていた。しかしながら、一般的に基本ブロックのサイズは大きくない。マルチスレッドで実行する場合、スレッド制御のオーバヘッドを隠蔽する観点から各スレッドの実行するコードサイズは大きい方が望ましい。そこで図2のようなノードの意味の読み替えを行なうことで複数の基本ブロックをまとめてコードサイズを大きくすることが可能となる。

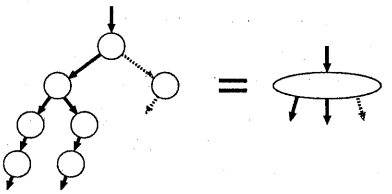

図2 基本ブロックの統合

この場合、各ノードから出る辺は2本以上になるが、実行確率の高い方から2つのみを考慮の対象とし、それ以外は各スレッドがそれぞれの実行を完了した後、継続して実行することで対応する。

またこの考えを応用することで多方向分岐も以上の枠組みに組み入れることができる。

2.1 スレッド制御命令

ここまでに述べた実行モデルを実現するためのスレッ

ド制御命令を以下に定義する。

- **fork** *id,limit,start_addr*
- **term**
- **commit** *id*
- **kill** *id*

fork 命令は子スレッドの生成を行なう。子スレッドの実行は投機状態で開始される。*id* は 0 または 1 の値をとり、それぞれメイン側かサブ側かを示す。*start_addr* は投機スレッドの実行開始点を示す。*limit* は投機実行を許すスレッドの最大数を示す。

term 命令は自スレッドの終了を行なう。**term** 命令実行時に自スレッドが投機状態であった場合は確定状態に移行するまで待機する。確定状態であった場合はスレッドの実行を終了し、プロセッサはアイドル状態となる。そこでもし次のスレッド生成要求があればそのスレッドの実行を開始する。

commit 命令は投機状態にある子スレッドを確定状態へ移行させる。この命令は自スレッドの実行の結果、子スレッドが実行されるべきであると確定した際に子スレッドに対して用いる。*id* によって 2 つある子スレッドのどちらのスレッドを確定状態へ移行させるかを指定する。*id* は **fork** 命令と同様に 0,1 の値をとり、それぞれがメインパス側、サブパス側を示す。

commit 命令は現在の実行状態が確定状態になった時のみ実行され、投機状態の場合は確定状態に移行するまで待機する。

kill 命令は子スレッドの実行の強制終了を行なう。この命令は自スレッドの実行の結果、子スレッドが本来実行すべきでないと確定した場合に使用する。*id* によってどちらの子スレッドを終了させるかを指定する。*id* は **fork**,**commit** 命令と同様に 0,1 をとり、それぞれがメイン側、サブ側のスレッドを指定する。

kill 命令は **commit** 命令とは違い、実行状態が投機状態であるか確定状態であるかによらず実行可能である。

kill 命令によって実行を無効化されたスレッドはもし自分に子スレッドがいればその実行を無効化する。

2.2 スレッド間通信

スレッド間の通信は親子関係のあるもの同志間のみで行なわれ、レジスタ間で行なうものとメモリアクセスによって行なうものの 2 種類が存在する。

レジスタ通信は以下の命令によって明示的に行なう。これは不必要な依存チェックを無くすためである。

- **send** *id,regno*
- **recv** *regno*

id は前出の **fork** 命令と同じものである。**send** 命令はレジスタ番号 *regno* の内容を *id*(0 または 1) で示されるスレッドの同番号のレジスタに転送する。各レジスタは親スレッドから内容が転送済みかどうかの情報を保持しており、**recv** 命令はレジスタ番号 *regno* に対応する状態チェックを行なう。親スレッドからのデータが到着していればそのまま後続の命令実行を続け、到着していなければ到着するまで待機する。

メモリアクセスに関して投機的なアクセスはその結果が確定状態になるまで外部のメモリシステムにその状態を反映してはならない。レジスタアクセスとは事情が異なり、メモリアクセスにはポインタ参照が存在するため、静的に全ての依存関係を追跡することが不可能である。そこでレジスタアクセスの場合とは異なった手法でスレッド間のデータ依存関係を動的に把握する必要がある。

本稿ではロードストアバッファをプロセッサとメモリシステムの間に存在することを仮定する。このロードストアバッファが親子関係にあるスレッドと通信を行ないながら全ての投機的なメモリアクセスを追跡し、プロセッサから発行された投機的なメモリアクセスがスレッド間の依存を保っているかどうかを自動的に判定を行なう。

3. スレッド分割

3.1 分 割 法

本複数パスモデルを実際のプログラムに適用する際のスレッド分割をどのように行なうかを以下に述べる。

プログラムから本モデルを使ったマルチスレッドコードへの分解は、各基本ブロックにおける累積した分岐確率を静的に計算することで行なう。

プログラムの制御構造を基本ブロックをノード、分岐方向を辺とした木構造で表現したものを調べる。各ノードを起点としてその子ノードにおける分岐確率を累積して計算を行ない、メイン側とサブ側のそれぞれの子孫のノードの実行確率を算出する。その結果に従って **fork** 命令の limit 値を計算する。

図 3に limit 値の計算の様子を示す。ここで、あるノードにおけるサブ側の limit 値 **R** を決めると自動的にそのメイン側の limit 値 **L** の最低値が決定される。これを L_{min} とおくと、これは $f(R)$ と表現できる。メイン側とサブ側のスレッドの総数は $L_{min}+R$ 以上になる。そこで、**L** と **R** の和が全プロセッサ台数を越えないもので最大の値となるような **R** の値を見つける。つまり以下の式を満す R を見つければ良い。(ただし N

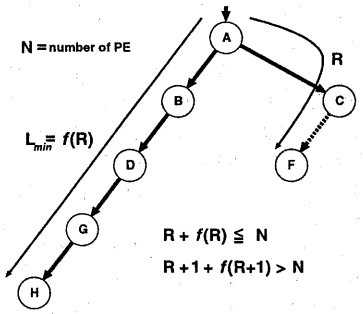

図 3 limit 値の算出

は全プロセッサ台数とする)

$$L_{min} + R = f(R) + R \leq N$$
$$L'_{min} + R + 1 = f(R+1) + R + 1 > N$$

Rが決定した後、Lの値は全プロセッサ台数NからRを引いた値にすれば良い。

以上の方法を実際にアプリケーションプログラムに適用した例を以下に述べる。今回は入力データによりプログラムの制御フローが大きく変化する特徴を持つプログラムとして二分探索プログラム (**bsearch**) とクイックソートプログラム (**qsort**) の2つを適用の対象とした。

3.2 BinarySearch

図4の左に二分探索の主要ループの制御構造を示す。二分探索では読み出した値によって次に読み出すべき値の位置が決定される。図に示す通り、このプログラムでは最初のブロックから2方向に分岐した後、それぞれループから脱出するかどうかを判定し、次のイテレーションに突入する。ここで最初のブロックから後続のそれぞれのブロックへの分岐は入力されたデータ値に依存するため、それぞれのブロックへの分岐確率は同等であるものと仮定することができる。

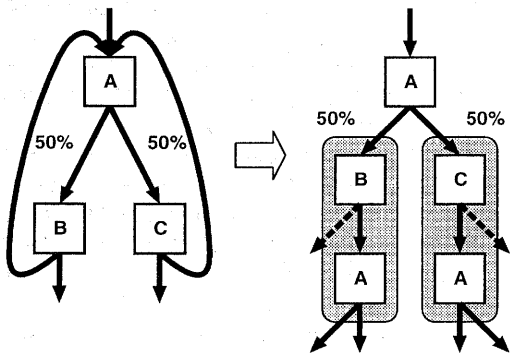

図4 binary search のスレッド分割

ブロックAを基点とした場合、分岐確率がメイン側サブ側双方同等であるため各limit値はどちらも同じ値になる。そこでこの場合の各limit値はプロセッサ台数の半分の値に設定する。ブロックBまたはブロックCを基点とした場合、ループの先頭に戻る方向とループから脱出する方向の確率を比較すると、圧倒的に前者の方が大きいと考えられる。そのため、メインをループ先頭へ戻る方向に、サブを脱出する方向に設定するとサブ側のlimit値Rが1以上の値になりえるのはメイン側のlimit値が極めて大きくなった時であるため、サブ側のスレッド実行は実質無い (つまり$R = 0$) ものとすることができる。そこでブロックB,Cに関してはメイン側のみ考えれば良い。

ここで図中の各ブロックA,B,Cをそれぞれ別のスレッドとして実行するものとし、それぞれのブロックの実行開始点に可能な限り近い地点に子スレッドを生成する命令を付加することが最も簡単な方法である。しかしこの場合、各ブロックのサイズが小さいため、スレッド生成消滅のオーバヘッドの分、性能が低下する可能性が大きい。そこで隣接するブロックをまとめて1つの複合ブロックとすることでブロックサイズを大きくし、スレッド制御のオーバヘッドによる影響を隠蔽する。

隣接するブロックをまとめる場合、ブロックAを先にブロックB,Cを後という順にするのが素直な方法である。しかし、この場合、ブロックB,Cで更新した値を次のブロックAで使用するためにそこで依存が生じ、投機実行がうまく働かない。そこで図4の右に示す通り、ブロックAを後にした形で複合ブロックを形成する。

結果として、ブロックB,Cの実行開始点に可能な限り近い地点でそれぞれ次のイテレーションのブロックB,Cを別の投機スレッドとして起動し、ブロックAの実行が完了する付近でどちらの子スレッドを確定して生かすかを判断する形のマルチスレッドコードとなる。

3.3 QuickSort

クイックソートの最内ループには図5の左に示す通りの制御構造が存在する。これは現在の基準値と入力配列の値との比較を行ない、基準値より大きい (あるいは小さい) 場所を特定する部分のコードに相当する。ループボディであるブロックAとブロックBはそれぞれ現在比較している入力配列の値により次のイテレーションに突入するかどうかが決定される。ここで各ブロックA,Bにおいてそれぞれからの分岐の方向は入力されたデータ値に依存するため、ループが脱出する方向への分岐確率とループを繰り返す方向への分岐確率は同等であるという仮定を置くことができる。

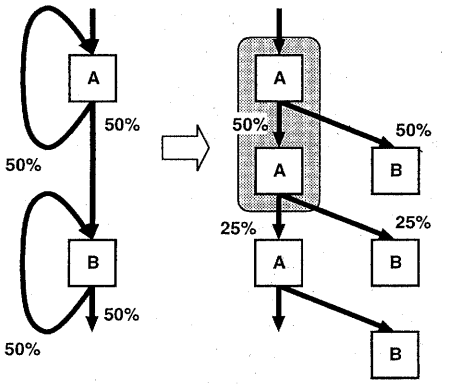

図5 quick sort のスレッド分割

図5において、左の制御フロー図は右の木構造と等価である。(ここで右図ではブロックBに関する部分を省略した)

この場合、ブロックA(あるいはB)の実行開始直後の早い時期にブロックA(あるいはB)を子スレッドとして生成することで、ループの各イテレーションを別個のスレッドとして投機実行を行なうことが最も単純なマルチスレッド化の方法である。

しかしながら、二分木探索の場合と同様に、クイックソートにおいてもループの各イテレーションのコードサイズ (つまりブロック A や B のサイズ) が小さいため、そのままではスレッドの制御にかかるオーバーヘッドの影響が出る可能性が大きい。そこでクイックソートの場合も隣接するブロックをまとめてブロックサイズを大きくする必要がある。

ここで 2 イテレーションを 1 つのスレッドとして実行するスレッド分割を考える。この場合、図 5の右に示す通りに 2 イテレーションをまとめてできた 1 つのスレッド (図中網掛け部分) からは 3 方向への分岐が存在することになる。各分岐の確率を同等であるという仮定の下では 3 方向のそれぞれへ分岐する確率は図に示す通り、1 イテレーション終了後脱出する分岐が 50 %、2 イテレーション終了後脱出する分岐が 25 %、次のイテレーションに突入する分岐が 25 %ということになる。

そこで分岐確率の大きいものから 2 つ、つまり 1 または 2 イテレーション終了後に脱出するコードをそれぞれメイン側サブ側のスレッドとして実行するものとする。

各 limit 値に関してはメイン側とサブ側の比が 2 対 1 となるように設定する。これはメイン側が 50 %、サブ側が 25 %であり、サブ側がメイン側の丁度二乗の値となっているため、メイン側の長さがサブ側の 2 倍になる地点が累積した分岐確率が一致する地点となるからである。

結果として、ブロック A の実行開始付近でそれぞれ 1 イテレーション実行後にループを脱出、2 イテレーション実行後にループを脱出に対応するスレッドを起動し、自分自信は 2 イテレーション分の実行を行ない、実行が完了する付近でその結果に応じてどちらの子スレッドを確定して生かすかあるいは双方の実行を破棄するかを判断する形のマルチスレッドコードとなる。ここで子スレッドのどちらかに実行が引き継がれる場合は自スレッドは終了するが、子スレッドの双方が破棄された場合は自スレッドが引き続き実行を継続することになる。

4. 性能評価

4.1 シミュレーション環境

前節で説明した 2 つのプログラムを用いてシミュレーションを行ない、性能評価を行なった。

シミュレータは直接バイナリコードを解釈実行を行なう命令駆動型のものをベースにマルチスレッド制御のための機能拡張を行なったものを用いた。バイナリコードは GNU C Compiler を用いて生成する。生成されたバイナリを基にしてスレッド制御を行なうための情報を外部に付加する。スレッド制御情報は基となるバイナリに制御関係の命令を直接埋め込む形ではなく完全に外部の情報として独立しており、オリジナルのバイナリに手を加える必要はなくなっている。

スレッド制御情報はアドレスと操作の対になっており、シミュレータはこのアドレスと実行中の命令アドレスとを比較し、一致した場合に現在のアドレスにある通常命令を実行する前にスレッド関係の操作を行なうような仕組みになっている。スレッド制御情報の分離することで、同じバイナリに対して複数のスレッド制御方法が実装できることが可能となっている。

シミュレータは 1 命令 1 サイクルで実行を行なう。全てのメモリアクセスはキャッシュに完全ヒットとしている。投機メモリアクセスに関しては 1 サイクルで全ての依存チェックが完了するものとしている。以下の結果において複数パス実行でのスレッドの制御にかかる命令数は単一パス実行の 2 倍になるため、制御にかかる命令オーバヘッドも 2 倍として処理している。

4.2 シミュレーション結果

図 6,7にプロセッサ台数とスレッドの制御にかかるコストを変化させた時の性能向上率を示す。図 6が二分木探索プログラムの場合、図 7がクイックソートプログラムの場合の結果である。それぞれの図において、プロセッサ台数を 4、8、16、32 台、スレッドの制御にかかるコストを 1、2、5 サイクルと変化させた場合に単一パス実行を行なった場合と複数パス実行を行なった場合の性能向上率を示している。図中の **SP** が単一パス実行を行なった場合、**MP** が複数パス実行を行なった場合である。ここで、性能向上率は 1 台のプロセッサで逐次実行を行なった場合にかかる実行時間を基準とした。

4.2.1 二分木探索

二分木探索プログラムにおいて、スレッドの制御にかかるコストがを 1 サイクルの場合、プロセッサ台数が 4 の時では単一パス実行では 5 %の性能向上であるのに対し、複数パス実行では 13 %向上した。台数を 8 にした時では単一パスで 6 %、複数パスで 26 %性能が向上した。しかしながら台数を 16 台以上にしても性能はそれ

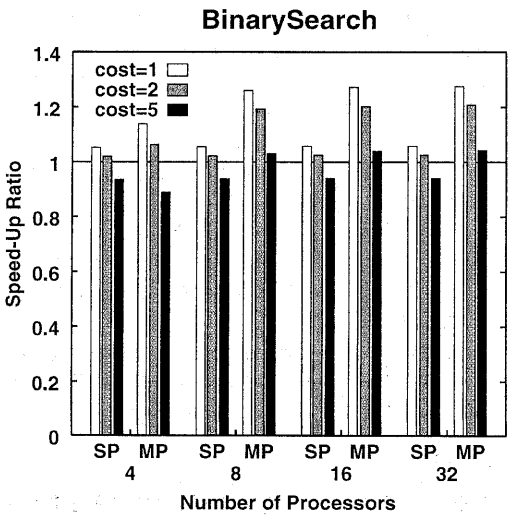

図 6 性能向上率 (binary search)

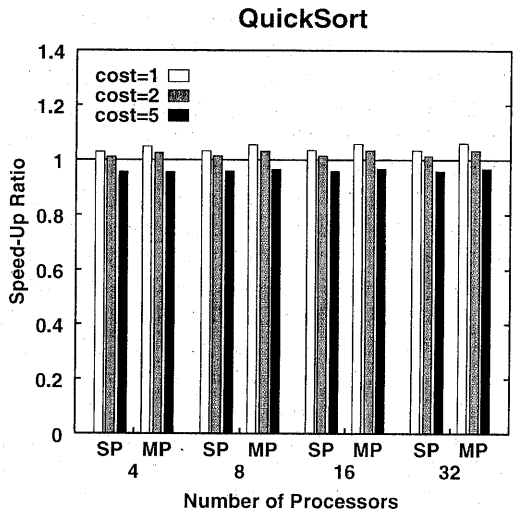

図7 性能向上率 (quick sort)

以上ほとんど上がらず、単一パスで6%、複数パスで27%である。

これはプログラム中の分岐が等しくどの方向にも分岐する可能性があり、その中から一つのパスを選んで投機実行を行なっても、先行して投機実行すればするほど急激に実際にその結果が使われる確率が減少していくため、プロセッサを増やしても有効な実行がほとんど増えないからである。

4.2.2 クイックソート

クイックソートプログラムにおいて、スレッド制御のコストが1サイクルの場合、プロセッサ台数をいくつに設定しても性能向上はわずかであり、単一パス実行の場合で3%、複数パス実行の場合で6%である。

これはスレッド制御のために実行される命令分のオーバヘッドが原因であると考えられる。二分木探索では2個の子スレッドのどちらかは必ず投機実行が成功するため、親スレッドの子スレッド制御にかかるオーバヘッドによる実行時間への影響は表われないが、クイックソートの場合は2個の子スレッドのどちらとも実行が破棄された場合に以後の実行を親スレッドが引き継ぐため、スレッド制御命令分のオーバヘッドがそのまま実行時間に加わることになる。実際にクイックソートで2個の子スレッドが全て破棄される確率を調べてみると、全体の15%は双方の投機実行が破棄されている。元のブロックサイズが小さいことから投機実行が成功した際の性能利得が小さいこともあり、投機が失敗した場合の影響が大きく出てしまったものと考えられる。

5. おわりに

投機的にマルチスレッド実行を行なう際に実行確率が大きいパス上のコードだけでなく、実行確率の小さいパス上のコードも選択的に投機実行の対象とすることで計算資源の有効利用を図り、全体の実行速度を向上させる複数パスモデルを実際のアプリケーションにどう適用するかを検討した。その適用例を用いて、複数パスモデルの有効性を検証するためにシミュレーションによる評価を行なった。その結果、単一パスモデルでは高速化が困難である場合に複数パスモデルを適用することで高速化が可能となる局面を確認できた。

謝辞 本研究において有効かつ適切なアドバイスを頂きました(株)東芝研究開発センター情報通信システム研究所 小柳滋氏に感謝致します。本研究は文部省科学研究費 基盤研究(B)課題番号10558039, 並列・分散処理研究推進機構の援助による。

参考文献

1) Wall, D.W.: Limits of Instruction-Level Parallelism, Technical Report DEC-WRL-93-6, Digital Equipment Corporation, Western Research Lab (93).
2) Lam, M.S. and Wilson, R.P.: Limits of Control Flow on Parallelism, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 46–57 (1992).
3) Sohi, G. S., Breach, S. E. and Vijaykumar, T. N.: Multiscalar Processors, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 414–425 (1995).
4) 鳥居淳, 近藤真己, 本村真人, 西直樹, 小長谷明彦: On Chip Multiprocessor 指向制御並列アーキテクチャ MUSCAT の提案, *Joint Symposium on Parallel Processing 97*, pp. 229–236 (1997).
5) 小林良太郎, 岩田充晃, 安藤秀樹, 島田俊夫: 非数値計算プログラムのスレッド間命令レベル並列を利用するプロセッサ・アーキテクチャ SKY, *Joint Symposium on Parallel Processing 98*, pp. 87–94 (1998).
6) Uht, A.K. and Sindagi, V.: Disjoint Eager Execution: An Optimal Form of Speculative Execution, *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 313–325 (1995).
7) 大津金光, 古川文人, 吉永務, 馬場敬信: 投機的マルチスレッド処理の一手法, 計算機アーキテクチャ研究会報告, pp. 61–66 (1998).
8) Ootsu, K., Yoshinari, W., Furukawa, F., Yoshinaga, T. and Baba, T.: A Speculative Multithreading with Selective Multi-Path Execution, *Proc. Innovative Architecture for Future Generation High-Performance Processors and Systems* (to be published).