2007－ARC－174　(36)
2007／8／3

実行時の分岐のふるまいに基づくスレッド間データ依存関係予測

中島貴裕[†]　菅原豊[†]　平木敬[†]

スレッドレベル投機実行において，投機スレッド生成と，失敗時の投機スレッドの破棄にはにはレジスタの値のコピーなどに伴うオーバーヘッドがかかる．そのため，結果的に投機失敗となる投機スレッドの生成，破棄を減らすことが性能向上につながる．投機実行の結果依存関係違反があった場合には，それ以降同じ地点で投機スレッドをを実行するにあたって投機スレッドの生成を中止する方法が過去に提案されている．しかし，この方法では時間経過とともにプログラム中の投機実行できる部分が単調に減少してしまい，性能向上も抑えられてしまう．本稿ではこの問題を解決するための，一度失敗予測の出た部分を，実行時情報を用いることによって投機実行する部分に復帰させるアルゴリズムと，シミュレーションによる性能評価について述べる．

Predicting Inter-Thread Data Dependencies Based on Dynamic Branch Behavior

Takahiro Nakajima, Yutaka Sugawara and Kei Hiraki

Spawning and squashing speculative thread takes additional machine cycles because it involves copy of register and pipeline filling. Therefore, preventing spawning speculative threads that are eventually squashed improves performance. A previous study has proposed a misspeculation predicting method. To prevent misspeculation, when misspeculation occurs, spawning this speculative thread is prevented next time. However, following this approach, once a misspeculation is predicted, there is no chance to spawn the speculative thread again. To solve this problem, we propose a method to collect information necessary to determine data dependency violation without spawning any speculative threads. We evaluated the performance using a simulator. The result shows that performance improvement can be achieved compared to the existing method.

1. はじめに

スーパースカラプロセッサは，プログラム中の命令レベル並列性を利用することで性能向上を果たしてきた．しかし，単一スレッド実行における命令レベル抽出には限界があり，その性能の限界を超えるためにスレッドレベル並列性の抽出が広く研究されている．スレッドレベル並列性を抽出するための１つの方法として，スレッドレベル投機実行がある．スレッドレベル投機実行は，プログラムをハードウェアレベルのスレッドに動的に分割し，プログラム内で将来実行されると予想されるスレッドを現在実行中のスレッドと並列に実行するものである．実行中のスレッド間に実行時の依存関係がなく，かつ投機スレッドに十分な粒度があれば性能が向上しうることが既存研究によりわかっている[3)6)]．

しかし，投機スレッドの操作にはオーバーヘッドが存在する．そのため，常に可能な場所で投機スレッドを発行すると，そのオーバーヘッドのために性能向上が妨げられる[8)]．そのため，最終的に失敗する投機スレッドの発行を抑えることができれば，性能が向上する．

それまでの実行結果に基づいて投機実行が成功するかを予測し，その結果によって選択的に投機スレッドの発行を行うという方法は既に提案されている[8)]．この方式では，投機実行の失敗が一定回数出た場合には投機状態から非投機状態へ移行し，非投機状態になったら次からは投機実行を抑制することで，無駄な投機スレッドの発行数を抑えるものである．

この予測に基づく方法の問題点は，一度投機失敗と予測された場合には二度と投機実行をしないので，投機実行が開始できる場所が単調に減少してしまうことである．

この問題を解決するために，本論文では投機スレッドを実際に実行しないときに，投機実行が成功していたかどうかの判定のための一方法を与える Revival Speculation Prediction を提案する．この方式を用いることで，非投機状態から投機状態への復帰の機会を

† 東京大学大学院情報理工学系研究科

```
main(){
    work_a;
    f()
    work_b;
    work_c;
}
```

```
f(){
    work_f;
}
```

Original Source Code

図 1 元のプログラムのコード.a,f,b,c の順で実行される.

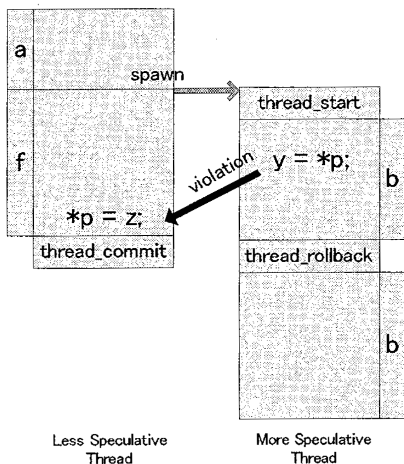

図 2 スレッドレベル投機実行. 並列に動作する 2 つのスレッドがあり,f と b を並列に実行しようとする.b と f に依存関係がある場合には, 投機スレッドは破棄され再実行する

与えることができる. 本稿の構成を述べる. 2 節では本稿でテーマとするスレッドレベル投機実行のモデルについて説明する. その上で既存の方法について述べ,問題点を明確化する. その上で 3 節では, その問題点を解決するアルゴリズムを提案し, 実現方法についても述べる. 4 節で性能評価を行う. 5 節で関連研究について述べ,6 節で本稿をまとめる.

2. スレッドレベル投機実行

本稿でテーマにするスレッドレベル投機実行のモデルは図 2 である. また, 投機スレッドの発行点は, 各関数の開始点とし, 投機実行する部分は関数の戻り先のプログラムとする. この投機実行モデルで理想的にはどれくらいの性能向上があるかは既に議論されている[6)7)]. 図 1 は元のプログラムのコードであり a, f, b, c の順で実行される.

a 部分の実行の後に, プログラムの制御が f の開始点に到達したときに投機スレッドが発行され, f と b が並列に実行される. この図において, 左側のスレッドは右側のスレッドに対して, プログラムをシーケンシャルに実行した場合には左側が先に実行される. よって, 左側で書込んだ値は右側のスレッドへフォワーディングされる.

2 つのスレッド間に依存関係がある場合には, 投機実行は失敗する. 図 2 では, 共有変数として p がある. 先に投機スレッドのほうで p の値を読み込んだあとで, フォワーディングの結果非投機スレッドで p への書き込みが起こった場合には, 先に読み込んでいた値が間違っていたことになるので, そのままだと正しいプログラムの実行にならない. よって投機スレッドは破棄され再実行される.

本稿でテーマにするモデルでは, スレッド管理のオーバーヘッドは, 図 2 上の `thread_start` と書かれた部分, つまり投機スレッド発行時にかかるペナルティと, `thread_rollback` と書かれた部分にあるとする. 前者はスレッド間のレジスタの値のコピーと, パイプラインが埋まるまでの時間であり, 後者は再実行の準備のための時間と考える.

これらのオーバーヘッドが全くない場合は性能は向上する. しかし, 投機実行における性能向上をこのオーバーヘッドが上回った場合には, 性能が低下する場合もありうる. 4 節における実験でそのような場合があることを述べる.

2.1 投機失敗予測

実行履歴を用いた投機失敗予測は既に提案されている[8)]. この方法は, ある場所から投機スレッドを発行して失敗した場合に, 次に同じ場所で投機スレッドを発行した場合のも失敗する可能性が高いという性質を利用している. この方式を実現するためにハードウェア資源として Violation Prediction Table(VPT) を追加する. VPT は投機スレッドの発行点をアドレスとして引くことができ, VPT 内には投機実行の履歴を保存される. 投機実行の開始のたびに, VPT をチェックし, 成功判定が出ていれば, 通常通り投機実行する. ある地点から開始した投機実行が失敗した場合には, 開始地点に対応する VPT を更新し, 次に VPT を参照した場合に失敗判定を出すようにする.

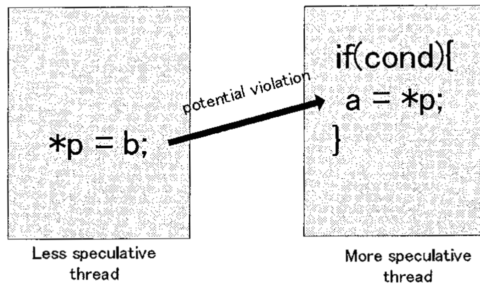

図 3 制御の流れに起因する投機成功の有無

2.2 既存手法の限界

それまでの投機の成功の可否によって投機実行をするかどうか判断する方法の問題点は，ある投機スレッド発行地点が一度非投機状態になった場合の動作にある．失敗予測が立っている以上，その地点から投機スレッドが発行されることはない．その結果，二度とその地点に対応する VPT が更新されることはなくなってしまう．プログラムの開始から時間が経つにつれ，投機状態にあるスレッド発行点は単調に減少してしまい，性能向上も抑えられてしまう．そこで一度非投機状態になった後で，もし投機スレッドを発行していたらその投機実行が成功していたかどうかを，プログラムの順序どおりに通常実行しながら判断する方法が必要となる．

3. 提案手法

この節では，投機スレッド発行点が非投機状態にあるときに，投機スレッドの発行をせずに投機状態に復帰する方法を提示する．

3.1 Revival Speculation Prediction

アルゴリズムは以下のものである．

- 投機失敗の原因となった命令をマークする
- マークした命令を監視する
- マークされた命令が実行されなくなったら投機再開

今回仮定しているものが関数の投機実行であるために，依存関係の違反はデータフローのみである．つまり違反が起こるのは，共有変数への書き込みと読み込み命令のみである．

今回提案する手法では，投機実行が成功するかどうかが制御の流れに起因していると仮定する．つまり，実行時の分岐のふるまいによって投機実行が成功するかどうかが変わるというものである．図 3 にそのような場合の例を示す．左側が非投機スレッドで右側が投機スレッドである．2 つのスレッド間の共有変数は p である．この場合，if 文の条件が成立した場合にのみ，依存関係違反が起こる．プログラム中の同じ部分は実行中に何度も実行するが，この if 文内の条件は常に同じではなく，変動する．このふるまいをとらえるために，この違反を起こす命令に到達したかどうかを投機復活の基準にする．

3.2 実現方法

本稿の提案のアルゴリズムの実現方法を提示する．

まず，グローバルな到達バッファを用意する．このバッファはプログラム内の命令アドレスによって引かれる．個々のエントリは 1 ビットで構成されており，0 ならば通過していなく，1 ならば通過したということである．命令実行中には，投機実行中かそうでないかにかかわらず，現在実行中の命令アドレスのエントリを 1 にする．以降，命令 P への到達状況として Reached[P] と表記する．

さらに VPT を拡張する．図 4 が VPT を拡張である．これまでの投機失敗の履歴に加えて，依存関係違反を起こした命令アドレスの対 P1,P2 を追加する．

プログラムの実行順に説明する．最初の段階では投機成功予測が出ているために，投機スレッドの発行は必ず行われる．投機失敗が起こった時点で，命令のアドレスの対が記録される．失敗が続くと，投機失敗予測が出るようになる．最初に投機失敗予測が出たときには，命令 P1 と命令 P2 は到達済みであるため，投機実行を行わない．しかし，Reached[P1] と Reached[P2] の更新はこれ以降で行われるために，次に発行地点に来た場合には状況が変化し，Reached の値が更新される可能性がある．その後で，失敗予測が出ている部分にきたとき，対応する命令 P1,P2 の到達状況をチェックする．Reached[P1] と Reached[P2] の両方の値が 1 の場合には，次に投機スレッドを発行しても失敗するとみなし，失敗予測を出す．Reached[P1] と Reached[P2] の値のどちらかが 0 だった場合は，次に投機スレッドを発行したなら成功するとみなし，成功予測を出す．なお，Reached[P1] と Reached[P2] の値は参照後は 0 にしておく．

この方法は 3.1 章のアルゴリズムを完全に実現するものではなく近似を与えるものであり，かつ予測結果は完全なものではないが，投機失敗状態から投機成功状態への復帰の機会を与えている．次章で性能評価を行う．

4. 性能評価

本章では，提案手法のシミュレータ上での性能評価を行う．まずは実行モデル，ベンチマーク，シミュレー

History	P1	P2

図 4　本稿で提案する VPT の拡張. P1 と P2 は前回依存関係違反を起こした命令ポインタの値.

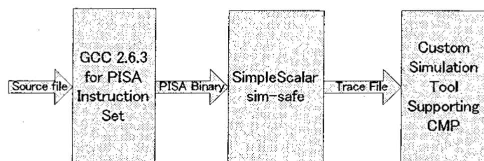

図 5　今回の性能評価のためのツールチェイン

タについて述べる.

4.1 実験に用いたパラメータ

CMP の各プロセッサエレメントはそれぞれシングルイシューのインオーダ実行をするものとする. また, 各プロセッサは理想的なパイプライン実行を行うものとする. つまり, 全ての命令のレイテンシは 1 であるとする. 1 つのチップ内のプロセッサエレメント数は 4 とする.

4.2 シミュレーション

今回の実行形態としてはトレースレベルシミュレーションを用いた. 図 5 が今回のシミュレーションのプログラムのツールチェインである. まず, 各ソースファイルは SimpleScalar 用にカスタマイズされた GCC 2.6.3 によって PISA 命令セットにコンパイルされる. 次に, それらのバイナリファイルは, SimpleScalar[1] の functional simulator である sim-safe によって実行される. これは我々の手によって改造されたもので, 解析に必要な実行トレースを出力する. この出力された実行トレースを, CMP による投機実行をサポートした我々のシミュレーションツールにより解析した. シミュレータは実際に 1 ステップずつ命令を実行する.

4.3 ベンチマーク

6 に今回用いたベンチマークアプリケーションをまとめる. 全てのベンチマークは SPEC CINT を使用した.

4.4 実行性能

図 7 の縦軸は PE 数が 4 のときの, 単一プロセッサでの実行に対する性能比である.

投機失敗予測を全く使わない場合には,099.go, 132.ijpeg, 164.gzip, 176.gcc で性能向上が見られるが,129.compress, 130.li and 197.parser では投機失敗のペナルティにより全く投機実行をしない場合よりも性能が低下している.

履歴を用いる単純な投機失敗予測を用いると, 予測

Name	Description
099.go	The Game of Go
129.compress	Unix Compress
130.li	Lisp Interpreter
132.ijpeg	JPEG Encoder/Decoder
164.gzip	GNU Zip Data Compression
176.gcc	GNU C Compiler
181.mcf	Combinational Optimization
197.parser	Syntactic Parser of English
255.vortex	Database Transaction

図 6　ベンチマークアプリケーション. 全てのプログラムは SPEC CINT からのものである.

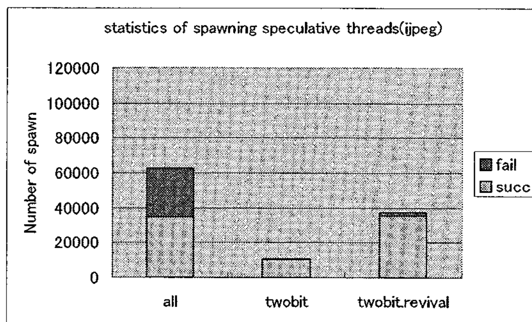

図 8　132.ijpeg における投機実行の. succ は成功した投機スレッドの数であり, fail は失敗した投機スレッドの数である

なしに比べてほとんどの場合性能が改善している. しかし,099.go と 132.ijpeg では予測なしの場合よりも性能向上は抑えられている. これは投機失敗数とともに, 投機成功数も減っているためである.

本論文の提案手法を用いた場合, 投機なしの場合にくらべて全てのアプリケーションで性能改善が見られる.132.ijpeg では 3 つの方式のうちで最もよい結果が得られた. 図 8 が示すように, 132.ijpeg では本論文の提案手法を用いることによって, 成功する投機スレッドの数を保ちながら失敗する投機を減らすことができたためだと考えられる.

197.parser では, 提案手法よりも既存の手法のほうがよい値を示している.

図 10 は投機スレッドのマネジメントに必要なサイクル数と実行性能の関係である.

投機スレッドのマネジメントに必要なサイクルがが 10 と 20 の場合には, 提案手法と既存手法の差が現れるが,50 サイクルになるとほとんど差が出ていない. この結果によると, オーバーヘッドが大きすぎる場合には効果が上がらないことがわかる. もしオーバーヘッ

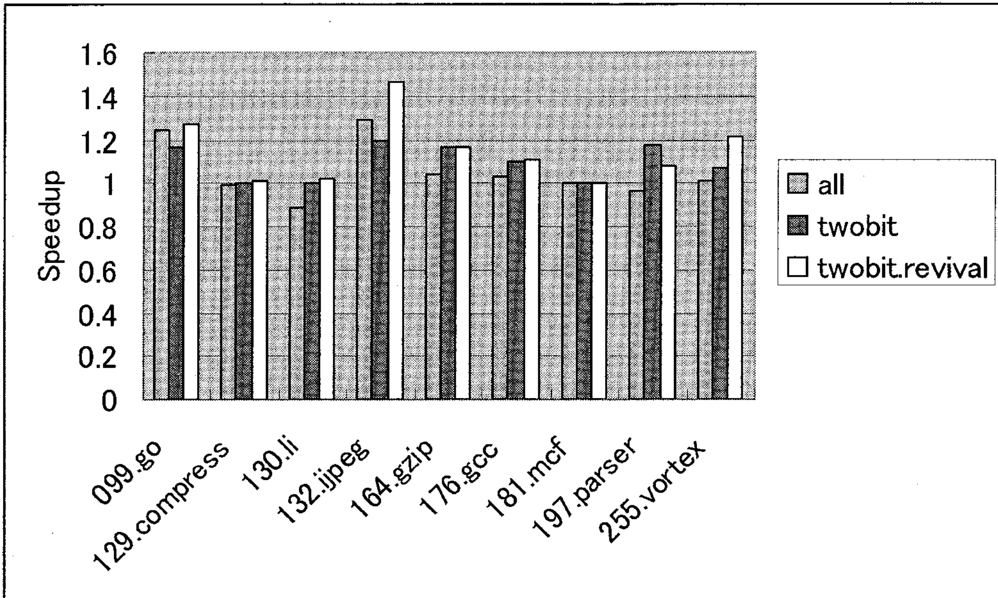

図 7　PE 数が 4 であり，オーバーヘッドが 20 の場合の，完全に順番に実行した場合に対する性能比.all が予測をせずに投機実行を行った場合で,twobit が 2 節で述べた既存研究[8]であり,twobit.revival が本稿における提案手法

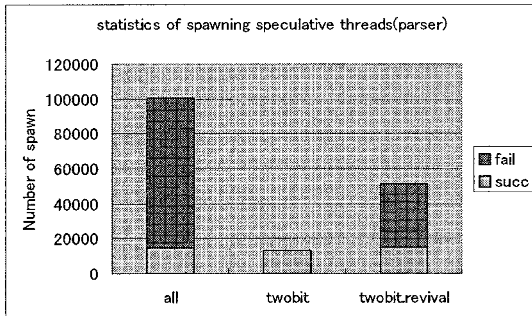

図 9　197.parser における投機実行の結果.succ は成功した投機スレッドの数であり，fail は失敗した投機スレッドの数である

ドが大きすぎる場合には，成功する投機実行が必ずしも性能向上につながらないためである．

5. 関 連 研 究

スレッドレベル投機実行が最初に提案されたのは Multiscalar[3] である．

本研究では，関数呼び出し点を投機スレッドの開始点としたが，その他投機スレッドの開始点の関しての研究が存在する[6]．

[2] では，投機スレッドの失敗を減らすために動的予測を行う方式を提案している．こちらの方式では各キャッシュラインに対して状態を持たせ，投機状態にあるデータのコミットを遅らせる方法であり，発行する投機スレッド自体の結果を予測するものではなく，その点で本研究とは異なっている．

[4] ではプログラムの実行トレースを用いることによって最適な投機スレッドの発行点を探している．

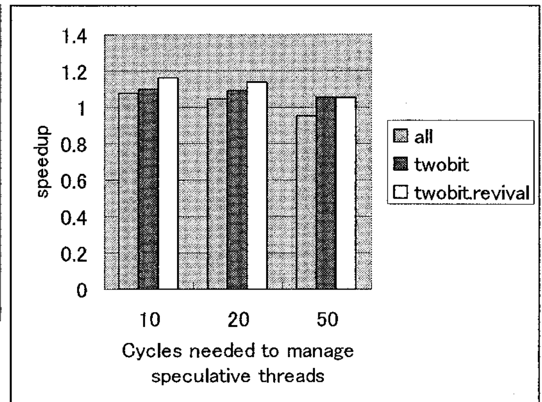

図 10　. 縦軸は投機なしで実行した場合に対する性能比. 値は全ベンチマークの幾何平均

実行時情報を用いることによってスレッドレベル投機実行の性能改善を行う方式には[5)7)]などがある．これらは投機スレッドの粒度を決定するために実行時情報を用いている．

6. まとめ

失敗する投機スレッドの数を減らすための従来のアルゴリズムでは，投機スレッドを開始できる場所が時間の経過とともに単調減少するという問題点を持つ．この問題点を解決するために依存関係違反を起こす命令の実行状態を監視するRSPという方式を提案し，シミュレーションによって性能評価を行った．その結果，従来方式に比べて性能向上が確認できた．

参考文献

1) D.Burger and T.M. Austin. The simplescalar tool set version 2.0. *University of Wisconsin-Madison Computer Sciences Department Technical Report (1997)*, 1997.
2) Marcelo Cintra and Josep Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture(HPCA'02)*, 2002.
3) Gurindar S.Sohi, Scott E.Breach and T. N.Vijaykumar. Multiscalar processors. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.
4) Michael K.Chen and Kunle Olukotun. Test: A tracer for extracting speculative threads. *Proceedings of Code Generation and Optimization, 2003*, 2003.
5) Niko Demus Barli, Daisuke Tashiro, Shuichi Sakai and Hidehiko Tanaka. Dynamic Thraed Extension for Speculative Multithreading Architectures. *Proceedings of Summer United Workshops on Parallel, Distributed and Cooperative Processing(SWoPP 2001)*, 2001.
6) Pedro Marcuello, Antonio Gonzalez and Jordi Tubella. Thread partitioning and value prediction for exploiting speculative thread-level parallelism. *IEEE Transaction on Computers, Vol 53, No.2*, 2004.
7) Fredrik Warg and Per Stenstrom. Improving speculative thread-level parallelism through module run-length prediction. *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.
8) Fredrik Warg and Per Stenstrom. Reducing misspeculation overhead for module-level speculative execution. *Proceedings of the 2005 ACM International Conference on Computing Frontiers(CF 2005)*, 2005.