Switch-on-Future-Event マルチスレッディングの改良と評価

倉田成己[†] 塩谷亮太[††]
五島正裕[†] 坂井修一[†]

シングル・プログラムの性能向上を妨げる大きな要因の 1 つとして delinquent 命令がある．delinquent 命令 とは，分岐予測ミスやキャッシュ・ミスなどのミス・ペナルティを頻繁に発生させる少数の静的な命令であり，またその大部分は比較的小さなループ中で実行されている．この delinquent 命令に対する手法としてヘルパースレッディングがあるが，これはプロセッサの資源を消費する．そのため，ミス・ペナルティが減少しているにも関わらず性能向上が最大でも 5.2%にとどまることが評価によりわかった．一方，我々の提案するループをマルチスレッド実行する手法ではそのような資源の消費が起きないため，より大きな性能向上が期待できる．さらにフェッチポリシを改良し，これまで理想化していたキャッシュのログを現実的な容量とした状態での評価で平均 10.5%，最大で 33.5%の性能向上となることを確認した．

Improvement and Evaluation of Switch-on-Future-Event Multithreading

NARUKI KURATA,[†] RYOTA SHIOYA,[††] MASAHIRO GOSHIMA[†]
and SHUICHI SAKAI [†]

Delinquent instructions are one of the largest causes of degrading performance of a single program. Delinquent instructions are a few static instructions that cause many branch prediction misses or cache misses, most of which are executed relatively in small loops. Helper threading is one of the existing techniques to hide latency of delinquent instructions, but this technique consumes extra resources of the processor. Therefore, although miss penalties have decreased, it only achieves performance improvement at a maximum of 5.2%. On the other hand, our latency hiding method of delinquent instructions by multithread execution of a loop does not consume the resources of the processor in such a way, so we can promise better performance improvement. In this paper, we propose improved instruction fetch policy of the method, and logs of speculative memory accesses in a realistic way, which we have been idealized. Simulation results shows that our proposal achieves performance improvement by an average of 10.5% and a maximum of 33.5%.

1. はじめに

シングル・プログラムの性能向上を妨げる大きな要因の 1 つとして，分岐予測ミスやキャッシュ・ミスなどのミス・ペナルティを頻繁に発生させる命令がある．本稿ではこうした静的な命令を **delinquent 命令**と呼ぶことにする．

この delinquent 命令のミス・ペナルティの遅延を隠蔽する手法として，**ヘルパースレッディング**がある[1)~4)]．ヘルパースレッディングでは，シングル・プログラムから delinquent 命令に関わる命令のみを抜き出してヘルパースレッドと呼ばれるスレッドを作成する．ヘルパースレッドをメインのスレッドに対して先行実行することにより，プリフェッチや分岐予測精度向上を図る．

ヘルパースレッディングでは，ヘルパースレッドとして抜き出す命令が少ないとメインのスレッドに対して十分に先行できず，逆に抜き出す命令が多いとヘルパースレッドがメインスレッドの実行を妨げ，性能低下を起こしてしまう．そのためヘルパースレッドの作成では最適な数の命令を抜き出す必要がある．

しかし，小さなループ中など delinquent 命令間の距離が小さい場合には，ヘルパースレッド自体の本数が増えるためヘルパースレッド 1 本あたりの命令数を減らしても全体の命令数は多くなる．すると，ヘルパースレッドは十分に先行できない割にメインスレッドの実行を妨げることになり，性能低下を引き起こす原因となる．

我々は，ループのサイズと，ループ中で実行された delinquent 命令の関係について評価を行った[5),6)]．**図 1** は分岐予測ミスについて，**図 2** は L2 キャッシュ・ミスについてのグラフである．グラフの横軸はミスを起こした命令が所属しているループのサイズを表す．ま

† 東京大学 大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†† 現在，名古屋大学 大学院 工学研究科
Graduate School of Engineering, Nagoya University

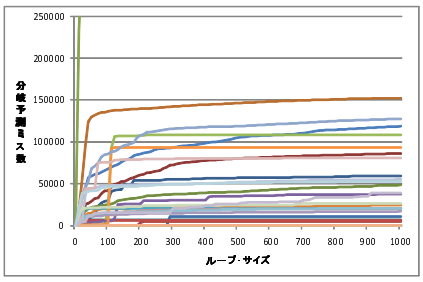

図 1 ループ・サイズ毎の累積分岐予測ミス回数

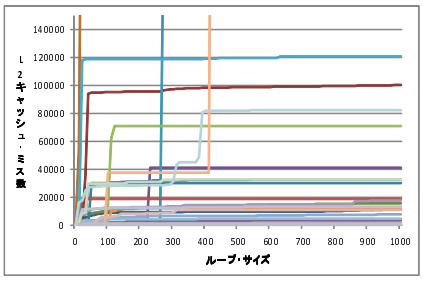

図 2 ループ・サイズ毎の累積 L2 キャッシュ・ミス回数

た，縦軸はそのサイズ以下のループで発生しているミス数の累積を表す．

図 1 より，分岐予測ミス数の絶対値が大きなプログラムでは，およそ 50 命令から 200 命令程度以下のループ内でミス回数が飽和していることがわかる．図 2 についても同様であり，L2 キャッシュ・ミス数の絶対値が大きなプログラムでは，多くの場合 400 命令以内のループでミス回数が飽和していることがわかる．

多くの delinquent 命令がこのような小さなループ中にあるため，ヘルパースレッディングではその効果を十分に発揮することができない．後の評価で述べるように，ヘルパースレッドが実行する命令は，メインスレッドで実行される命令に対して最大で約 32%にもなる．

我々は，あるスレッドの実行中にフェッチした命令が delinquent であると予測されると，別のスレッドに切り替える **Switch-on-Future-Event マルチスレッディング**（SoF-MT）を提案した[6),7)]．SoF-MT では，delinquent 命令の実行が完了するまで別スレッドを実行することにより，その遅延を隠蔽する．この手法ではヘルパースレッドのような余分な命令を実行するのではないため，プロセッサ資源が余分に消費されることがない．我々は特に多くの delinquent 命令がループ中で実行されていることに着目し，ループの各イタレーションをスレッドとしてマルチスレッディングを行うこととした．

本稿では，SoF-MT のスレッド切り替えポリシを改良することにより，より効果的な遅延の隠蔽を行う手法を提案する．また，これまでの SoF-MT のモデルでは投機的なメモリ・アクセスに関して理想化を行っており，無限大の容量をもつログを用いて評価を行っていた．本稿では，投機バッファという現実的な容量のバッファを用いて評価を行い，ヘルパースレッドに対し十分に小さなハードウェア資源でも SoF-MT が有効に機能することを示す．

本稿の構成は以下の通りである．まず 2 章で，ヘルパースレッディングとその問題点についてまとめ，3 章で SoF-MT について述べる．次に，4 章で SoF-MT の改良について説明する．5 章で提案手法の定量的な評価を行った後，6 章では Switch-on-Future-Event マルチスレッディングの関連研究について述べる．

2. ヘルパースレッディング

ヘルパースレッディングとは，ヘルパースレッドと呼ばれるスレッドをメインのスレッドに対して先行実行することによって性能を向上させる手法である[1)~4)]．このヘルパースレッドは，通常，シングル・プログラムから delinquent 命令に関わる命令のみを動的に抜き出して作成される．以下ではこのヘルパースレッディングについて説明し，その問題点について述べる．

2.1 ヘルパースレッディングの実行手順

ヘルパースレッドがメインスレッドに対して十分先行するためには，ヘルパースレッドの命令数がメインスレッドの命令数よりも十分に少なくなければならない．また，ヘルパースレッドの実行はプロセッサ資源を消費するため，ヘルパースレッドの命令数が多いとメインスレッドの実行を妨げてしまう．このため，ヘルパースレッディングでは，ヘルパースレッドの作成方法が重要なポイントとなる．

ヘルパースレッドは，まず delinquent 命令の検出を行い，それが依存する命令のみを抜き出すことで作成する．以下ではまず delinquent 命令の検出方法について説明し，次にヘルパースレッドを作成する手法，更にスレッドのトリガ方法について述べる．

delinquent 命令の検出

delinquent 命令を動的に検出する方法については，ヘルパースレッドを提案している文献では詳しく述べられていないが，ここでは我々が過去に提案した **DIT**(Delinquent Instruction Table) を用いて行うことにする[6)]．DIT は命令アドレスをインデックスとしてアクセスするタグ付きの飽和型カウンタになっており，分岐予測ミスやロードミス数を数えている．この DIT が一定の閾値を越えていた場合，その命令を delinquent 命令として検出する．DIT についての詳細な動作については 3.3 節で説明する．

ヘルパースレッドの作成

ヘルパースレッドの作成には，あらかじめリタイアした命令を保存しておくバッファを用意する．命令列の作成方法としては，まず命令のリタイア時に DIT にアクセスして delinquent 命令かどうか確認する．その結果 delinquent であることが検出されると，その依存

元の命令をさかのぼって再帰的に加えていくことにより動的にヘルパースレッドを作成する[1]．

スレッドのトリガ

ヘルパースレッドは，その先頭の命令に対するプロデューサ命令によってトリガされ，実行が開始される．

2.2 ヘルパースレッディングの問題点

ヘルパースレッディングでは，ヘルパースレッドとして抜き出す命令が少ないとメインのスレッドに対して十分に先行できず，逆に抜き出す命令が多いとヘルパースレッドがメインスレッドの実行を妨げ，性能低下を起こしてしまう．そのためヘルパースレッドの作成では最適な数の命令を抜き出す必要がある．

しかし，小さなループ中など delinquent 命令間の距離が小さい場合には，ヘルパースレッド自体の本数が増えるためヘルパースレッド 1 本あたりの命令数を減らしても全体の命令数は多くなる．すると，ヘルパースレッドは十分に先行できない割にメインスレッドの実行を妨げることになり，性能低下を引き起こす原因となる．1 章で示したように delinquent 命令は小さなループ中にあるため，ヘルパースレッディングではその効果を十分に発揮することができない．

ヘルパースレッドがメインのスレッドに対してどの程度さかのぼるかはヘルパースレッドの先行度とプロセッサ資源の消費のバランスを考慮しなければならない重要な要素であり，5 章でその評価を行う．

3. Switch-on-Future-Event マルチスレッディング

我々は delinquent 命令に対し，シングル・プログラム中のループから生成したスレッドを同時に実行することで遅延を隠蔽する Switch-on-Future-Event マルチスレッディング (SoF-MT) を提案している[6]．SoF-MT はヘルパースレッディングのように命令を余分に実行するのではないため，上記のようなヘルパースレッディングの問題は発生しない．以下ではまず，単一プログラム内のループをマルチスレッド実行する方法を述べた後，delinquent 命令の遅延を隠蔽する手法について述べる．

3.1 マルチスレッド実行のモデルと遅延の隠蔽

SoF-MT では，ループの各イタレーションをスレッドとし，SMT と同様の物理構成を持つプロセッサ上でそれらのマルチスレッド実行を行う．**図 3** に示すように，プロセッサ上の各スレッドを論理プロセッサとして，各論理プロセッサが単方向で接続されているようなモデルを考える．

例として**図 4** に，for ループ中に delinquent な分岐命令が存在した場合の遅延の隠蔽の様子を示す．図中の $if([i])$ が delinquent 命令とする．今，スレッド t_0 の分岐命令をフェッチし，delinquent と判定されたら次のスレッド t_1 にフェッチ先を切り替えて実行を継続

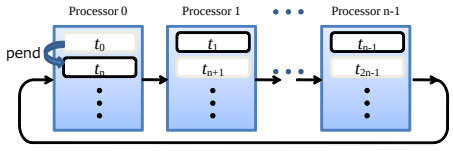

図 3 n 個の論理プロセッサへのスレッドの割り付け

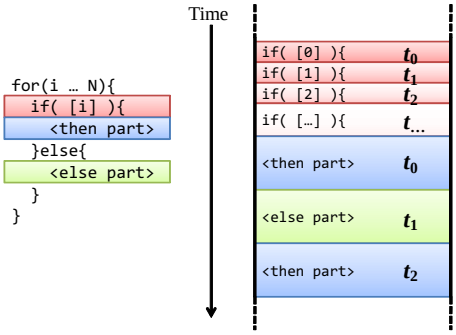

図 4 if-then-else 構造を持つループの実行

する．このスレッドでも，delinquent 命令をフェッチしたら次のスレッドにフェッチ先を切り替える……という動作を繰り返し，t_0 の分岐先が決定したところでフェッチ先を元に戻すと，予測をすることなく分岐の遅延が隠蔽される．また，delinquent 命令がロード命令であった場合でも同様で，命令をフェッチしたときにフェッチ先のスレッドを切り替えることで，ロード命令に依存する後続の命令が長時間命令ウィンドウに滞留するのを防ぐことができる．

3.2 プログラムに挿入する専用命令

SoF-MT のマルチスレッディング実行開始や終了，レジスタ繰り越し依存の解決を行うため，いくつかの専用命令をループ中に挿入する．以下ではそれらの命令について説明する．

3.2.1 マルチスレッディングの開始と終了

スレッドの実行開始と終了は，いくつかの専用命令をループに挿入しそれらを実行することによって行われる．

- **pstart**　この命令が実行されると，マルチスレッド実行が開始され，n 個の論理プロセッサが存在した場合 0 から $n-1$ 番目までのイタレーションに対応するスレッドが生成される．
- **pend**　スレッドの完了を表し，pend の完了時に，それまで実行を行っていた論理プロセッサに対しまだ割り当てが行われていない中で最も先行するイタレーションを割り当てる．また pend は先行する全てのイタレーションが終了するまで実行されない．
- **pexit**　マルチスレッドの実行を終了する．ループを脱出し，後続の命令の実行を再開する．

これらの命令はプログラマが明示的に指定するか，実行時プロファイルの情報を元にコンパイラによって挿入される．

3.2.2 繰り越し依存の解決

レジスタのイタレーション間における繰り越し依存の解決は，各論理プロセッサに**レシーブバッファ**と呼ぶエントリを用意し，**send** と **recv** と呼ぶ専用の命令によって行う．レシーブバッファはプロセッサのレジスタの数だけ用意する．また，論理プロセッサ i は論理プロセッサ $(i+1)\%n$（% はモジュロ演算子）に対してのみに値を送信する．この send と recv の挿入は前述の専用命令と同様にコンパイラによって静的に行われる．send による値の送信は非投機的に行われ，send の完了時に値が送信される．

recv の中で，ループのイタレーションごとに単調な変化をする変数に関しては単純な予測器で値予測が可能なため，先行スレッドの send によって値を受け取る前に投機的に実行を継続することができる．予測にはストライド値予測器[8] を用いる，ストライド値予測器は予測する recv のデスティネーションレジスタをタグとする CAM で構成されており，直近 2 回の recv した値の差分と，最近の値，信頼度カウンタをエントリとして持つテーブルになっている．予測時には信頼度カウンタを参照し，MSB が 1 であれば最近の値＋差分値を予測値とする．閾値を超えないときやエントリが存在しないときは予測を行わず，値が send されるのを待つ．信頼度カウンタはここでは飽和型の 2bit カウンタとし，recv の実行完了時に差分値が前回と同じならインクリメント，違っていればデクリメントされる．予測の検証はレシーブバッファに予測値を残すことで行い，予測が間違っていればそのスレッドと後続のスレッドを再実行する．

3.3 delinquent 命令の検出

delinquent 命令には分岐命令とロード命令があり，分岐命令は **DIT** と呼ぶテーブルで delinquent 命令を検出する．また，長時間命令ウィンドウ中に滞留する recv 命令についても DIT で delinquent 命令として検出する．DIT の各エントリがタグ付きの飽和型カウンタであり，テーブルへのアクセスは命令アドレスの一部をインデックスとしてアクセスする．命令の実行完了時に分岐予測ミスや recv 命令の長時間の滞留があった場合，DIT の対応するエントリのカウンタをインクリメントする．エントリが存在しない場合は新たにエントリを確保し，0 に初期化する．分岐予測がヒットしたり，recv 命令が十分に短い時間で実行終了した場合には，対応するエントリが存在したときにカウンタをデクリメントする．delinquent 命令の検出は命令フェッチ時に行い，アドレスに対応するエントリを読み出しカウンタが閾値を超えていればその命令を delinquent 命令とする．インクリメントする値とデクリメントする値を変えることで，どの程度のヒット率で delinquent 命令とするかを調節することができる．例えばインクリメント数 2 に対してデクリメント数 1 としたとき，分岐予測器のヒット率が約 66%未満だと delinquent 命令とみなすことに相当する．

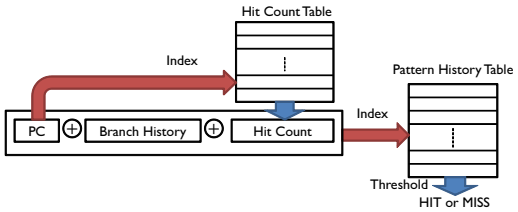

図 5 周期的なロード・ミスを予測する機構

ロード命令については，周期的なキャッシュ・ミスを予測できる機構を用いて delinquent 命令の検出を行う．ループ中ではイタレーションのを用いてアクセスするようなロード命令が多く存在し，その命令が周期的にキャッシュ・ミスするため，周期を予測できれば効果的なスレッドのスイッチが期待できる．

具体的には，**HCT**（Hit Count Table）と **PHT**（Pattern History Table）の 2 つのテーブルを用いて周期的なロード・ミスの予測を行う．**図 5** は周期的なキャッシュ・ミスを予測する仕組みを表している．HCT は命令アドレスをインデックスとしてアクセスする．各エントリはカウンタになっており，キャッシュの連続ヒット回数を保存する．また，PHT の各エントリは飽和型のカウンタで，テーブルへのアクセスは命令アドレスとグローバル分岐履歴，HCT から引いたヒットカウントの XOR をインデックスとする．ロード命令の実行完了時にキャッシュ・ミスが起きていた場合，HCT の対応するエントリを読み出し，その値を元に PHT のインデックスを計算して対応するエントリのカウンタをインクリメントした後，HCT の対応するエントリの値を 0 に初期化する．キャッシュ・ヒットの場合，HCT の対応するエントリを読み出し，その値を元に PHT のインデックスを計算して対応するエントリのカウンタをデクリメントした後，HCT の対応するエントリの値をインクリメントする．delinquent 命令の検出は，命令フェッチ時に HCT の対応するエントリにアクセスし，エントリが存在していれば PHT の対応エントリにアクセスする．読み出したエントリのカウンタが閾値を超えていればその命令を delinquent 命令として検出する．

3.4 実行結果の保証

SoF-MT におけるマルチスレッド実行ではループを逐次実行した場合と同じ結果を保証するため，通常は先行するイタレーションの実行が終了するまでは，それに依存する後続のイタレーションを実行することができない．このため，前述したスレッド切り替えを単純に実装した場合，このような実行不能な後続のイタレーションによって命令ウィンドウの実効エントリが少なくなったり，最悪の場合にはデッドロックが発生してしまうことがある．SoF-MT では，スレッド切り替えのポリシを工夫し，2 段コミットと呼ぶ機構を用いてこの問題を解決する．以下では最も先行するスレッ

ドを**非投機スレッド**，それ以外の後続のスレッドを**投機スレッド**と呼ぶ．

3.4.1 スレッド切り替えの制御

先行スレッドの命令がそれに依存する後続の命令よりも後にフェッチされてしまうのを避けるため，基本的には先行するスレッドを優先的にフェッチする．これを実現するため，いくつかのフラグとカウンタを追加する．まず，各論理プロセッサ毎に **WDF**（Wait Delinquent instruction Flag）と呼ぶフラグを保持する．WDF はその論理プロセッサが delinquent 命令の実行を待っていることを示す．論理プロセッサが delinquent 命令をフェッチすると WDF は 1 にセットされ，実行を終了すると 0 に戻される．さらに，各論理プロセッサ毎に **IN**（Iteration Number）と呼ぶカウンタを設ける．このカウンタはループを逐次実行した際のイタレーション実行順を表し，先行するスレッドから順に連続した一意の番号が割り当てられる．

SoF-MT では，各サイクルで WDF が 0 となっていて，IN が最も小さい論理プロセッサのスレッドをフェッチする．ただし，全ての論理プロセッサの WDF が 1 となっている場合は，IN が最も小さい論理プロセッサのスレッドをフェッチする．

3.4.2 2 段コミット

SoF-MT のマルチスレッド実行では，逐次実行と同じ結果になることを保証する．このため，ナイーブな実装を行った場合は先行する全てのスレッドの命令が完了するまで，後続のスレッドの命令は完了できない．したがって，最も先行するスレッド以外の後続のスレッドの命令は，実行が終了しているにも関わらず，命令ウィンドウのエントリを占有し続ける．これらの後続スレッドの命令による命令ウィンドウの占有は，命令ウィンドウの実効エントリ数を減らすばかりでなく，最悪の場合はプログラム進行のデッドロックを発生させる．そこで，提案手法では最も先行するスレッド以外のスレッドもコミットを行い，プロセッサからリタイアさせる．これを **1 段目のコミット**とする．また，後述のメモリ・アクセス・ログをメモリに反映しシステム全体の情報を確定することを **2 段目のコミット**と呼ぶ．

メモリ・アクセス順序違反の検出

SoF-MT ではプログラムを逐次実行した場合と同じ結果を保証するため，異なるスレッドが同一のアドレスにメモリ・アクセスをした場合は結果を巻き戻す必要がある．SoF-MT では，メモリ・アクセスした論理プロセッサの IN とメモリ・アドレス，データ，メモリ・アクセスがロードかストアか，の 4 項目をメモリ・アクセス・ログとして無限に保存できることとしている．論理プロセッサがメモリにアクセスするごとにログを無時間で検索し，別の論理プロセッサによる同一のメモリ・アドレスへのアクセスがあった場合にメモリ・アクセス順序違反を検出する．ただし，自身のアクセスがロードであり，かつ読み出したアクセス・ログが全てロードであった場合には，実行に矛盾は発生しないため順序違反とはしない．

順序違反が起きた場合，IN の大きい論理プロセッサの実行するスレッドとその後続のスレッドが再実行され，ログは消去される．また，論理プロセッサがスレッド，つまりイタレーションの実行を終了した際，その論理プロセッサに対応するストアのデータがキャッシュに書き込まれる．その後，ログが消去される（2 段目のコミット）．

4. SoF-MT の改良

SoF-MT では，スレッド切り替えのポリシを工夫し，2 段コミットと呼ぶ機構を用いて実行結果を保証した．しかし，従来の手法ではスレッド切り替えのタイミングが不適切な場合があり，またメモリ・アクセス順序違反の検出を理想化していた．本章では，この 2 点についての改良手法を提案する．まずフェッチ・ポリシを最適化し，スレッド間で依存する命令について配慮を行う．また，現実的な方法でのメモリ・アクセス順序違反の検出を行い，さらにメモリ・アクセス順序違反をする命令のメモリ・アクセスを遅延させることで，最適なタイミングで命令を実行する手法を提案する．

4.1 フェッチ・ポリシの最適化

フェッチを行うスレッドの決定は WDF と IN に加え，**WPF**（Wait Preceding thread Flag）と呼ぶフラグを用いる．WPF はその論理プロセッサが先行するスレッドに依存する命令をフェッチし，先行スレッドの対応する命令の完了を待っている状態を示す．論理プロセッサが先行するスレッドに依存する命令をフェッチすると WPF は 1 にセットされ，命令が先行スレッドの命令によってウェイクアップされると 0 に戻される．また，WPF は非投機スレッドでは 1 にセットされない．

まず，各サイクルで WDF と WPF がいずれも 0 となっている論理プロセッサのうち，IN が最も小さい論理プロセッサのスレッドをフェッチする．もし全論理プロセッサの WDF もしくは WPF が 1 となっている場合，非投機スレッド（IN が最も小さいスレッド）のフェッチを行う．

ここで，非投機スレッドは WPF がセットされないため，通常は WPF が立っているスレッドがフェッチされることはない．しかし，非投機スレッドがイタレーションの終了を示す pend をフェッチすると，そのスレッドにおいてそれ以上のフェッチが行われないため，次に先行しているイタレーションのフェッチをすることになるが，この論理プロセッサは WPF が立っていることがある．このとき，もし pend が前方の分岐命令の予測ミスによりフラッシュされ，さらに後続のスレッドへの繰り越しが完了していないと，後続スレッ

ドの pend 以降にフェッチした命令が滞留することになる．

そこで，全論理プロセッサにおいて WDF または WPF が 1 で，かつ非投機スレッドが pend をフェッチした場合は，WPF が 0 の論理プロセッサのうち最も IN の小さいスレッドのフェッチを行う．また，非投機スレッド以外の全論理プロセッサの WPF が 1 となっていた場合は，非投機スレッドの次に IN が小さい投機スレッドのフェッチを行うが，非投機スレッドの pend が分岐予測ミスによりフラッシュされた場合は，pend 以降にフェッチした後続スレッドの命令も同時にフラッシュすることによって命令の滞留を防ぐ．

スレッド間依存命令には recv 命令と，メモリ・アクセス順序違反を引き起こすロード・ストア命令がある．recv 命令に関しては 3.3 節 と同様に行い，recv 命令をフェッチした論理プロセッサの WDF ではなく WPF を 1 にする．

4.2 メモリ・アクセス順序違反の検出

SoF-MT ではプログラムを逐次実行した場合と同じ結果を保証するため，後続のスレッドがメモリ・アクセス順序違反を起こした場合に結果を巻き戻す必要がある．その方法として，Transactional Memory[9] とほぼ同様の機構を用いることが考えられる．1 次キャッシュの各ラインに論理プロセッサの数だけ SR と SW を追加し，1 段目のコミットによりメモリに対して投機的な Read ないしは Write が行われたことを示す．

このようなキャッシュ・ライン単位で各スレッドのメモリ・アクセスを管理する手法では false sharing が発生する．すると，そのラインに対して複数のスレッドが書き込みを行ったのでメモリ・アクセス順序違反とみなされ，スレッドの再実行が発生する．ループ中ではイタレーションを用いてメモリにアクセスすることが多く，このような false sharing による再実行が多発し，性能の低下を招く原因となる．

これを防ぐには，L1 キャッシュの各バイトごとにスレッドの数だけ SR と SW を追加することが考えられる．しかし，その場合はデータビット対して，SR と SW の割合が大きく増加してしまい，ハードウェアが大きくなるという問題が発生する．

そこで，新たに小容量のバッファ（**投機バッファ**）を用意する．投機バッファは各バイトごとに SR と SW，valid ビット， dirty ビットを追加した構成になっており，L1 キャッシュと並列にアクセスできるようになっている．また，投機バッファと L1 キャッシュはライン単位でデータをやりとりできるようになっており，ラインのデータの一部をマスクして書き込みが可能である．この投機バッファを L1 キャッシュと比較して十分小さくすることで，SR ビットや SW ビットによるハードウェアの増加を抑える．5 章では，投機バッファが L1 キャッシュと比較して少容量でも十分な性能が得られていることを示す．

投機バッファへのアクセス方法は非投機スレッドと投機スレッドで異なる．以下ではそれぞれに分けて説明する．

非投機スレッドの場合

非投機スレッドでは，投機バッファと L1 キャッシュを同時にアクセスする．投機バッファにラインが無かった場合は，通常のメモリ・アクセスと同様の動作をする．このとき，投機バッファにはデータやビットの書き込みを行わない．

投機バッファにラインが存在した場合は，投機バッファに対しても操作を行う．このとき，投機バッファの対応するバイトの SR と SW を参照し，それらが 1 であった場合にメモリ・アクセス順序違反として検出する．ただし，自身のアクセスが Read であり，読み出されるバイトに対応する SR のみが 1 であった場合にはコンフリクトとはしない．これは，同一アドレスに対する Read 同士では実行に矛盾が起きないためである．また，書き込みを行う際は dirty ビットを立てない．これは，投機バッファと同時に L1 キャッシュにも書き込みを行っているためである．

メモリ・アクセス順序違反を検出した場合は，違反した投機スレッドの結果を破棄し，そのスレッドと後続のスレッドを再実行する．再実行を行うスレッドに対応する SW が立っているバイトは全て無効化する．

投機スレッドの場合

投機スレッドにおいても，投機バッファと L1 キャッシュを同時にアクセスする．ただし，データやビットに対する操作は投機バッファに対してのみ行う．投機バッファにアクセスするバイトが一部でも無かった場合は，通常のキャッシュから投機バッファにリフィルを行い，このとき対応する SR または SW を立てる．また，valid となっているバイトには上書きしないようマスクをする．

リフィルを行った際に追い出されるラインのどこかに dirty ビットが立っていた場合は，dirty ビットが立っているバイト以外をマスクして L1 キャッシュに書き込む．また，SR または SW が立っていた場合は対応するスレッドの再実行を行う．

メモリ・アクセス順序違反を検出した場合は，違反したスレッドとその後続スレッドの結果を破棄し，再実行を行う．

4.3 メモリ・アクセス順序違反する命令の予測

メモリ・アクセス順序違反を起こした命令は，次回以降も違反を起こし，再実行が頻発する可能性が高い．これを防ぐナイーブな実装としては，過去にメモリ・アクセス違反を起こした命令をフェッチしたときに，先行するスレッドが全てコミットされるまでフェッチを止めることが考えられる．

しかし，順序違反を起こした命令と，本来先にアクセスすべき先行スレッドの命令とのペアがわかれば，先行スレッドの命令が実行を完了した時点で後続の命

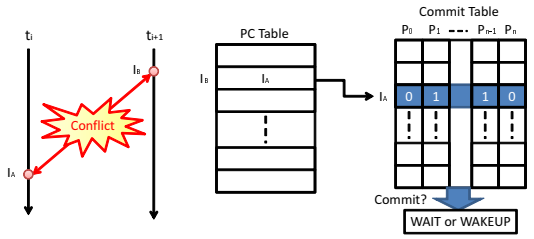

図 6 メモリ・アクセス順序違反する命令の予測

令をウェイクアップすることにより最適なタイミングで後続スレッドの実行を再開することが可能である．本手法では後続の命令をフェッチしたときに WPF を立て，ウェイクアップされるタイミングで WPF を下ろすことによりこれを実現する．

図 6 は，メモリ・アクセス順序違反する命令を予測する機構を表す．メモリ・アクセス順序違反を起こした命令ペアの検出には，**PCT**（PC Table）と **CMT**（Commit Table）を用いる[7]．PCT は命令アドレスをエントリとして持ち，命令アドレスまたはデータアドレスをインデックスおよびタグとしてアクセスを行う．また，CMT は論理プロセッサ数のビットをエントリとして持ち，命令アドレスをインデックスとしてアクセスするタグ付きのテーブルである．

命令ペアの作成は以下のとおりである．まず，先行するスレッドの命令が後続のスレッドのメモリ・アクセス順序違反を検出する．そこで，アクセスしたメモリのデータ・アドレスをインデックスとしたエントリを PCT に確保し，その中に先行するスレッドの命令アドレスを入れる．その後，後続のスレッドは再実行され，PCT のエントリにデータ・アドレスをインデックスとしてアクセスすることでペアとなる命令のメモリアドレスを発見できる．そこで，後続スレッドの命令の命令アドレスをインデックスとしてエントリを確保し，そこに発見した先行スレッドのメモリ・アドレスを入れる．これによって，後続スレッドの命令アドレスをインデックスとして PCT にアクセスすることにより，先行スレッドの命令アドレスを知ることができる．最後に，アクセスするメモリ・アドレスをインデックスとしたエントリを無効化する．

先行するスレッドの命令が実行完了したかどうかは，CMT によって確認する．まずスレッドが開始した際に，対応する論理プロセッサのビットをすべて下ろす．CMT にはメモリ・アクセス命令が実行完了時にその命令アドレスをインデックスとしてアクセスする．エントリが存在した場合は実行している論理プロセッサに対応するビットを立てる．メモリ・アクセスを待つ命令は，PCT を通して CMT にアクセスし，先行スレッドに対応する論理プロセッサのビットが立ったことを確認するまでウェイクアップされる．また，先行スレッドが実行完了したら，CMT のビットに関係なくウェイクアップされる．

表 1 プロセッサの構成

パラメータ	値
ISA	Alpha 21164A
logical thread	4 way
fetch width	4 inst.
execution unit	int : 2, fp : 2, mem : 2.
instruction window	int : 32, fp : 16, mem : 16
register file	int : 256, fp : 256
branch prediction	8KB g-share
miss penalty	10 cycle
BTB	2K entry, 4-way
L1C	32KB, 4-way, 64B/line, 2 cycle
L2C	4MB, 8-way, 64B/line, 10 cycle
main memory	100 cycle

表 2 評価に用いたベンチマーク

ベンチマークセット	アプリケーション
SPECCPU 2006[10]	perlbench, bzip2, mcf, milc, gromacs, hmmer h264ref, lbm, astar
MediaBench[11]	adpcm_dec, adpcm_enc
EEMBC[12]	dither

表 3 ヘルパースレッドのハードウェア構成

パラメータ	値
DIT	1KB, 4-way, 3 bits/count
Retired Insn. Buffer	1K entry

4.4 SoF-MT の改良点のまとめ

SoF-MT の改良点についてまとめると，まずこれまでの SoF-MT では，通常の delinquent 命令と先行スレッドに依存する命令を同一に管理していた．しかし，これを切り分けることでより適切なスレッドの切り替えを行うことが出来るようになった．

またこれまでの手法では理想化していたが，提案していたキャッシュ・ライン単位で各スレッドのメモリ・アクセスを管理する手法では false sharing が発生して再実行が多発し，性能の低下を招く原因とることがわかった．これをバイト単位で管理するバッファを用意することによって，より細かなメモリ・アクセスの管理が可能となった．

またそれに伴い，Switch-on-Future-Event マルチスレッディングの改良[7] で提案していたスレッド間で同一のメモリ・アドレスにアクセスする命令のペアを検出し，メモリ・アクセスを遅らせる手法にも変更を加えた．

5. 評　　価

プロセッサ・シミュレータ鬼切弐[13] に対し，以下のモデルを実装して評価を行った．評価したプロセッサのパラメータは**表 1** の通りである．また，ヘルパースレッドと SoF-MT 特有のハードウェア構成については，それぞれ**表 3** と**表 4** に示している．

- **Baseline**　通常のシングル・スレッド実行を行うモデル

表 4 Switch-on-Future-Event のハードウェア構成

パラメータ	値
DIT	1KB, 4-way, 3 bits/count
HCT	160B, 2-way
	8 bits/count, 2 bits/conf
PHT	256B, 2 bits/count
PC Table	256B, 2-way
Commit Table	16B, 2-way
Stride Value Predictor	32 entry, CAM

- **Helper**　マルチスレッディングによりヘルパースレッドを実行するモデル
- **SoF**　提案する SoF-MT のモデル

表 2 評価に用いたベンチマークをに示す．ただし，ヘルパースレッディングのパラメータ設定においては SPEC CPU2006[10] に含まれる全アプリケーション (29 本) についてのシミュレーションを行った．プログラムのコンパイルには gcc4.3.3 をベースとし，3 章で述べた専用命令を挿入することができるコンパイラを作成し用いた．ヘルパースレッドを測定するためのバイナリはこのコンパイラに専用命令を追加しないオプションを用いて作成した．コンパイラの最適化オプションは-O3 を使用する．MediaBench を除く各ベンチマークでは最初の 1G 命令をスキップし，続く 100M 命令を実行した． MediaBench についてはプログラムによる実行命令数が少ないため，全命令の実行を行った．なお我々が作成したコンパイラは現状プログラマの手でソースコードのループを指定することにより専用命令を挿入する仕様である．そのため，ベンチマークは delinquent 命令の特定が出来ており，かつループの指定が可能なものを選んでいる．

以下ではまずヘルパースレッディングのパラメータ設定のための評価を行い，次に設定したパラメータを用いたヘルパースレッディングと SoF-MT を評価する．

5.1 ヘルパースレッディングのパラメータ設定

ここでは，以下の 2 つのパラメータを変化させて設定を行った．なお，このパラメータ設定においては SPEC CPU2006 に含まれる全アプリケーション (29 本) についてのシミュレーションを行った．

- **最大命令数**　1 つのヘルパースレッドを構成する最大命令数．最大の命令数が多くなるほどヘルパースレッドが先行できるようになるが，より多くのプロセッサ資源を消費する．
- **DIT のインクリメント**　分岐予測ミス，キャッシュ・ミス時に DIT の値をいくつインクリメントするかの値．インクリメントする数が多いほど delinquent 命令を検出しやすくなるが，誤検出も増える．

SMT 構成に関しては 4 スレッドであり，メインスレッド 1 本に対しヘルパースレッドを最大 3 本同時に走らせることが可能とする．

図 7 は最大命令数を固定し，DIT のインクリメント

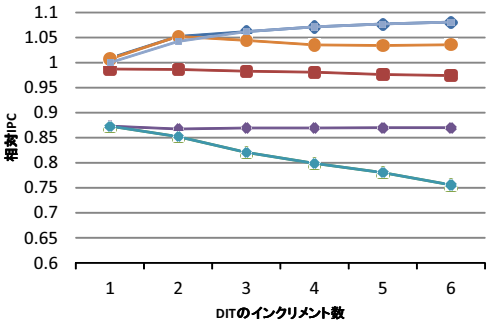

図 7　最大命令数を 16 としたとき

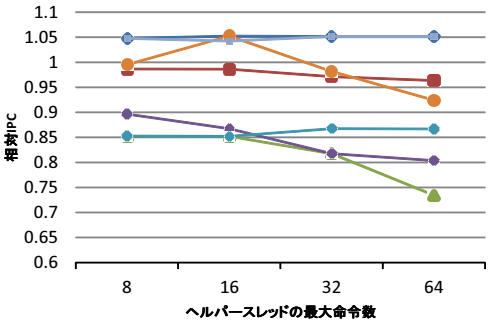

図 8　DIT のインクリメントを 2 としたとき

数を変化させたときの結果である．横軸は最大命令数，縦軸は Baseline モデルを基準とした相対 IPC で，各項目は全アプリケーションの最大，最小，平均値と，特徴的なベンチマーク 4 本である．この図より，DIT のインクリメント数を増やしていくごとに 433.milc は相対 IPC が増加しているが，性能向上率が 8%付近で飽和している．逆に 462.libquantum に関しては相対 IPC がほぼ線形に減少している．また 429.mcf はインクリメント数が 2 の時に最大となっているがそれ以外はほぼ横ばいで，435.gromacs においてはインクリメント数を変えてもあまり IPC に変化が見られない．また**図 8** は DIT のインクリメント数を固定し，最大命令数を変化させたグラフである．433.milc と 462.libquantum といった DIT のインクリメント数に対して変化するベンチマークはそれほど変化が見られないが，435.gromacs に関しては減少している．また，429.mcf はヘルパースレッドの最大命令数が 16 のときに最も性能が向上し，それ以上では性能が低下している．

このように，どのパラメータが最適であるかはアプリケーションによって変わるが，delinquent 命令の判定基準を緩く，ヘルパースレッドの最大命令数を多くすると，性能向上が頭打ちになり，逆に性能が低下するアプリケーションが増えることがわかる．これは，不必要にヘルパースレッドの実行を増やすと，プロセッサ資源を消費し，メインスレッドの実行を妨げているからだと考えられる．

以降では，ヘルパースレッディングにおける DIT の

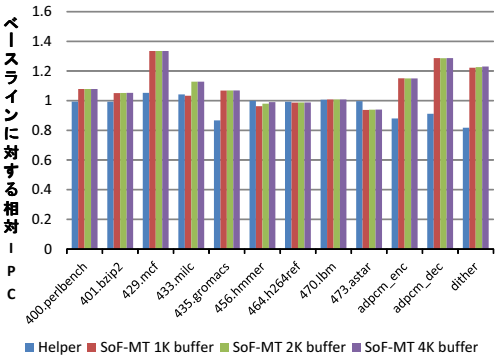

図 9 Baseline に対する相対 IPC

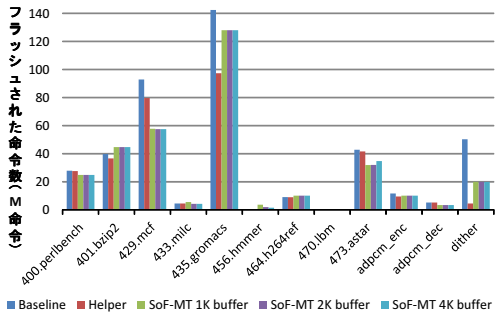

図 10 フラッシュされた命令数

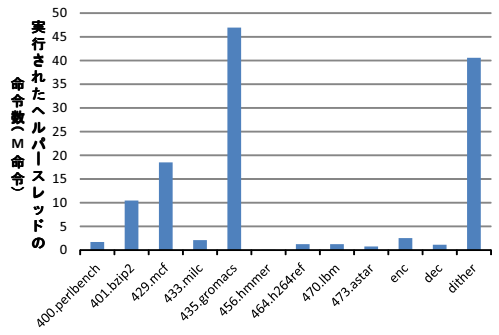

図 11 実行されたヘルパースレッドの命令数

インクリメント数は 2，最大命令数は 16 とする．

5.2 評価結果

図 9 はベースラインに対する相対 IPC，**図 10** は分岐予測ミスによってフラッシュされた命令数，**図 11** はヘルパースレッディングにおいて実行されたヘルパースレッドの総命令数である．「SoF-MT nK buffer」は，投機バッファの大きさを nKB としたときのデータを表す．

ヘルパースレッディングにおいて，いくつかのベンチマークで性能の向上がみられるが，最大でも mcf の 5.2%である．一方，SoF-MT では mcf で 33.5%の性能向上を達成している．

また gromacs に注目すると，ヘルパースレッディングでは分岐予測ミスによりフラッシュされた命令数が Baseline や SoF-MT よりも少ないにもかかわらず，相対 IPC は 13.3%も低い．これは図 11 を見るとわかるように，プログラム全体の約 32%という多くのヘルパースレッドの命令の実行にプロセッサ資源が消費され，メインスレッドの実行を妨げているからだと考えられる．一方 SoF-MT では，フラッシュされた命令数はヘルパースレッディングよりも多いものの，相対 IPC では 6.9%の性能向上を達成している．

また SoF-MT における投機バッファの容量別に見ていくと，1KB から 2KB に増やした際に milc で 9.5%の性能向上がみられる以外は，いずれも 2%未満の変化に留まっている．さらに今回評価した 12 種類のベンチマークのうち 9 種類が Baseline 以上の性能を達成しているため，ほとんどの場合投機バッファの容量は 1KB ないしは 2KB 程度で十分だと考えられる．

一方で，hmmer において 3.8%の性能低下が起きている．これは，投機バッファのリフィルが起きた際に追い出されたキャッシュ・ラインの SR または SW のビットが立っており，投機スレッドのアボートが多発していることが原因である．

6. 関連研究

SoF-MT に関連する手法として，Speculative Multithreading（SpMT）と SMT のフェッチポリシで分岐予測の信頼度を用いる手法を紹介する．

6.1 Speculative Multithreading

SpMT は，シングル・プログラムのスレッド・レベル並列性を利用して高速化を図る手法として古くから研究されており，代表的な例としては Multiscalar[14]，SKY[15]，Pinot[16] などが挙げられる．SpMT は Processor Element(PE) と呼ばれる演算ユニットを 4 個程度束ね，隣接する PE にレジスタ転送が可能な単方向リングで結んだ構成となっている．各 PE が制御等価な部分に分割されたプログラムを並列実行することで高速化を図る．制御等価とは，プログラムが基本ブロック X を通るときは必ずその後に基本ブロック Y を通り，かつ基本ブロック Y を通るときは必ず基本ブロック X を通るような組 (X, Y) のことである．

SpMT はシングル・プログラムの高速化を図る点では SoF-MT と同じであるが，delinquent な命令の遅延を隠蔽はせず，シングル・プログラムのスレッド・レベル並列性を利用する．

6.2 分岐予測の信頼度を利用した SMT のスレッドスイッチ

互いに独立なスレッドを実行する SMT プロセッサにおいて，分岐予測器の信頼度を利用してフェッチするスレッドを選択することによってスループットを向上させる研究がいくつか存在する．Luo らは，フロントエンドに存在する分岐命令に対する分岐予測器の信頼度が高いスレッドを優先してフェッチすることによ

り性能向上を図っている[17]．この手法では，予測器の信頼度が低い分岐命令を持つスレッドが投機的な命令を極力持たないようにすることで予測ミスによってフラッシュされる命令数を減らし，プロセッサ全体のスループット向上を狙っている．

こうした研究は，基本的には互いに独立な複数のプログラムを同時に実行することを前提としている．分岐予測器の信頼度が低いスレッドのフェッチを抑えることにより全体のスループットは向上するが，分岐予測が当たった場合を考えるとフェッチを抑えられたスレッド単体の性能は低下することになる．一方，本研究ではこうした分岐予測器の信頼度が低いプログラムの性能向上を目的としている点でこれらの研究とは異なる．またこれらの手法で単純にシングル・プログラムの遅延を隠蔽しようとしても，逐次実行と同様の結果を出すためには SoF-MT のようなスレッド切り替えのポリシやメモリ・アクセスに対する工夫が必要になってくる．

7. おわりに

本稿では delinquent 命令の遅延の隠蔽を目的とした手法として，delinquent 命令の先行実行を行うヘルパースレッディングとシングル・プログラム中のループをマルチスレッド実行する SoF-MT に関する議論を行った．SoF-MT のスレッドスイッチのポリシを改良し，また今まで理想化されていた投機的なメモリ・アクセスについての具体的な実装方法を示してヘルパースレッドとの比較を行った．その結果，ヘルパースレッディングが最大 5.2%の性能向上である一方，SoF-MT では現実的なハードウェアを用いて最大 33.5%の性能向上を達成できることがわかった．今後の課題としては，投機バッファから追い出されたラインに投機ビットが立っていて，アボートが多発することが挙げられる．これを防ぐために，投機バッファの追い出そうとしているキャッシュ・ラインに SR か SW が立っていた場合は，そのビットが降りるまで実行を待つなどの工夫を検討していきたい．

謝　辞

本論文の研究は一部， JST CREST，および，文部科学省科学研究費補助金 No. 23300013 による．

参 考 文 献

1) Collins, J., Wang, H., Tullsen, D., Hughes, C., Lee, Y.-F., Lavery, D. and Shen, J.: Speculative precomputation: long-range prefetching of delinquent loads, *ISCA*, pp. 14–25 (2001).
2) Collins, J., Tullsen, D., Wang, H. and Shen, J.: Dynamic speculative precomputation, *MICRO*, pp.306–317 (2001).
3) Roth, A. and Sohi, G.: Speculative data-driven multithreading, *HPCA*, pp. 37–48 (2001).
4) Chappell, R., Stark, J., S. Kim, S. R. and Patt, Y.: Simultaneous Subordinate Microthreading (SSMT), *ISCA*, pp. 186–195 (1999).
5) 塩谷亮太, 五島正裕, 坂井修一: 分岐プレディシジョン, 情報処理学会研究報告 2008-ARC-179, pp. 67–72 (2008).
6) 塩谷亮太, 倉田成己, 中島潤, 五島正裕, 坂井修一: Switch-On-Future-Event マルチスレッディング, 先進的計算基盤システムシンポジウム SACSIS2010, pp. 157–165 (2010).
7) 倉田成己, 塩谷亮太, 中島潤, 五島正裕, 坂井修一: Switch-On-Future-Event マルチスレッディングの改良, 情報処理学会研究報告 2010—ARC—190 (2010).
8) Wang, K. and Franklin, M.: Highly accurate data value prediction using hybrid predictors, *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, Washington, DC, USA, IEEE Computer Society, pp. 281–290 (1997).
9) Herlihy, M. and Moss, J. E. B.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *ISCA*, pp. 289–300 (1993).
10) The Standard Performance Evaluation Corporation: *SPEC CPU2006 suite http://www.spec.org/cpu2006/*.
11) Lee, C., Potkonjak, M. and Mangione-Smith, W.: MediaBench: a tool for evaluating and synthesizing multimedia and communications systems, *MICRO*, pp. 330–335 (1997).
12) The Embedded Microprocessor Benchmark Consortium: *http://www.eembc.org/*.
13) 塩谷亮太, 五島正裕, 坂井修一: プロセッサ・シミュレータ「鬼斬弐」の設計と実装, 先進的計算基盤システムシンポジウム SACSIS, pp. 120–121 (2009).
14) Sohi, G., Breach, S. and Vijaykumar, T.: Multiscalar processors, *ISCA*, pp. 414–425 (1995).
15) 小林良太郎, 小川行宏, 岩田充晃, 安藤秀樹, 島田俊夫: 非数値計算応用向けスレッド・レベル並列処理マルチプロセッサ・アーキテクチャSKY, 情報処理学会論文誌, Vol. 42, No. 2, pp. 349–366 (2001).
16) Ohsawa, T., Takagi, M., Kawahara, S. and Matsushita, S.: Pinot: Speculative Multi-threading Processor Architecture Exploiting Parallelism over a Wide Range of Granularities, *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, USA, IEEE Computer Society, pp. 81–92 (2005).
17) Luo, K., Franklin, M., Mukherjee, S. and Sezne, A.: Boosting SMT performance by speculation control, *International Parallel and Distributed Processing Symposium*, pp. 9 pp.– (2001).