Vol.2010-ARC-188 No.1
2010/3/1

マルチコアプロセッサにおける メモリ依存予測及び同期機構

秋田晃治[†1] 小林良太郎[†2] 安藤秀樹[†1]

メモリ依存に関するスレッド投機において，メモリ依存違反は性能向上の妨げとなる．本論文はメモリ依存違反を回避するためのメモリ依存予測/同期機構を提案する．本機構は過去の履歴に基づいて，静的なストアとロードのペアについてメモリ依存関係を予測する．そして，動的にデータ・アドレスごとの同期を行う．8 コアのマルチコアプロセッサにおいて，SPEC2000，Mediabench から選んだ 14 本のベンチマーク・プログラムを用いて，提案機構の評価を行った結果，提案機構はメモリ依存予測及び同期を行わなかった場合の性能に対して，7.6%の性能向上が得られた．

Memory Dependence Prediction and Synchronization on Multi-core processor

Koji Akita,[†1] Ryotaro Kobayashi[†2] and Hideki Ando[†1]

In the thread speculation on memory dependence, memory dependence violation degrades performance. This paper proposes a memory dependence prediction / syncronization mechanism to avoid memory dependence violation. This mechanism is based on the past history and predicts memory dependence between static store-load pairs. Then, it synchronizes them regarding data addresses dynamically. Our evaluation on the multi-core processor with 8 cores results using 14 programs from SPEC2000 and Mediabench benchmark suits shows that our mechanism achives a 7.6% speedup over the execution without prediction and synchronization.

†1 名古屋大学
Nagoya University
†2 豊橋技術科学大学
Toyohashi University of Technology

1. はじめに

命令レベル並列 (ILP : Instruction-Level Parallelism) の限界から，プロセッサの性能向上のためには，スレッド・レベル並列 (TLP : Thread-Level Parallelism) の利用が不可欠となっている．これまで TLP 利用に関する多くの研究がなされているが，大きく 2 つに分類できる．1 つは，マルチプログラミング環境でのスループット向上に関するものである．もう 1 つは，単一のプログラムの実行時間を短くする研究である[1),2)]．本論文では，後者に関し焦点を当てる．

マルチスレッド実行では，スレッドの並列実行によりプログラムの意味が変わらないようにするため，レジスタとメモリに関するスレッド間のデータ依存関係を見つけ出し，同期/通信を行う必要がある．このうち，レジスタに関してはコンパイラでの静的な解析が容易に行える．

一方，メモリに関しては静的な解析が容易であるとは限らない．特に，関数間依存や不規則なデータ・アクセスのパターン，非数値計算プログラムにおけるポインタによる参照などは解析が困難である．コンパイラは依存がわからなければ依存があると保守的に仮定しなければ，プログラムの意味を保つことができないため，TLP の利用が妨げられる．

そこで，ハードウェアの支援を得て，スレッドをメモリ依存に関して投機的に実行することにより，TLP を積極的に利用する研究がある[3),4)]．最も単純な方法は，メモリ依存が存在しないと楽観的に予測し実行するものである．実際に依存が存在しなければ投機は成功して，TLP を利用できる．しかし，メモリ依存が存在した場合，メモリ依存違反が生じ，投機は失敗して回復のペナルティを被る．

上述の方法は単純であるが，メモリ依存違反の頻度が高く，性能低下を引き起こす．このため，ハードウェアによる動的なメモリ依存予測及び同期により，メモリ依存違反を回避することが重要になる．

本論文ではハードウェアによる動的なメモリ依存予測及び同期を実現するための機構を提案する．これによって，投機失敗の頻度を減少させ，性能低下を回避する．

本論文では我々がこれまでに提案してきたマルチコアプロセッサ SKY[5),6)] を仮定して，評価する．SKY は，複数の独立したコアプロセッサを持ち，各コア間に高速なレジスタ通信機構を備えたアーキテクチャである．

© 2010 Information Processing Society of Japan

Vol.2010-ARC-188 No.1
2010/3/1

本論文は 2 章で SKY のマルチスレッド・モデルについて説明し，3 章でメモリ依存とその予測及び同期について説明する．4 章では提案するメモリ依存予測/同期機構について説明する．5 章で提案機構の評価を行う．6 章で関連研究について述べる．7 章でまとめを行う．

2. SKY のマルチスレッド・モデル

本研究で評価のベースとした SKY アーキテクチャにおけるマルチスレッド・モデルについて説明する．SKY のマルチスレッド・モデルは，スレッド並列実行のためのオーバーヘッドを小さくするため，通常のマルチスレッド・モデルと比べて次に示す制約を課している．

- 各スレッドは，逐次実行における動的に連続する部分で構成される
- 各スレッドは，逐次実行の順において自分の直後の部分をスレッドとして生成する

図 1 (a) の逐次実行命令列に対する SKY のスレッド分割を同図 (b) に示す．同図に示すように，SKY における各スレッドは，動的な命令列における単一の連続した部分からなり，異なる複数の部分からは構成されない．したがって，スレッドに結合はなく，制御に関する同期は必要ない．

図 1 (b) に示したスレッドを並列に実行する様子を，同図 (c) に示す．同図 (b) において，各スレッドに T_0，T_1，T_2 と逐次実行の順に名前をつける．同図 (c) に示すように，スレッド T_i は実行途中で，スレッド T_{i+1} を生成するという逐次生成を繰り返す．各スレッドは実行中には高々1 回しか新しいスレッドを生成しない．実行のできるだけ早い時期に新しいスレッドを生成することにより，スレッドの並列実行を実現する．スレッドの生成は `fork` と呼ぶ専用の命令を用い，終了は `finish` と呼ぶ専用の命令を用いて行う．

スレッドは実行を開始してから終了するまで，1 つのコアを占有する．あるコア上のスレッドは `fork` 命令を実行することによって，後続コアに子スレッドを生成する．ただし，後続コアが他のスレッドによって占有されているなら，`fork` 命令は無効化され，子スレッドは生成されない．この場合，スレッドに分割され実行されるべき部分は，当該スレッドによって逐次に実行され，プログラムの意味は保持される．

3. メモリ依存とその予測及び同期について

メモリ依存がどのように生じるかを説明し，依存の解決手法も同時に説明する．

3.1 依存を持つストアとロード

依存はプログラムの意味により生じるから，静的なストア命令とロード命令の対により安定して存在すると考えられる．これらの命令の存在は，メモリ依存違反が生じたときに判明する．

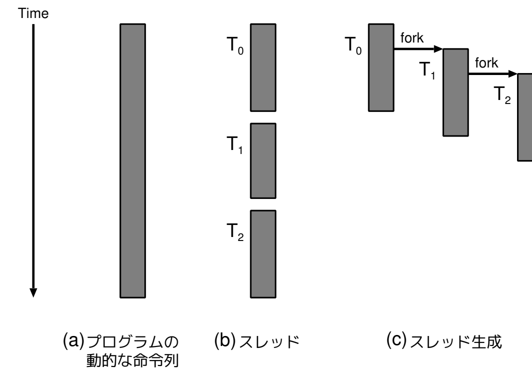

図 1 SKY のマルチスレッド・モデル

そこで，違反発生時に違反を起こした命令の PC を記録する．記録した PC によって，依存のある命令を識別することにより，メモリ依存を予測して，メモリ依存違反を回避する．

3.2 データ・アドレス

静的に依存を予測したとしても，あるロードのインスタンスがどのストアのインスタンスに依存するかを判断するには十分でない．依存関係はデータ・アドレスごとに動的に生じるから，データ・アドレスによってインスタンスを区別し，同期を行うとする．

メモリ依存があると予測されたロードは，ロードのデータ・アドレスと一致するストアの実行を待ち合わせ，同期を行う．

3.3 真の依存が生じるスレッド間隔

データ・アドレスごとの同期を行うとき，真の依存ではなく，誤って依存認識 (以下，偽の依存と呼ぶ) したストアがメモリ同期のトリガとなる場合がある．例えば，図 2 のスレッド 0 とスレッド 1 でそれぞれデータ・アドレス B に対しての書き込みがある．スレッド 0 の `i2` のストアによって同期をトリガされる可能性があるが，`i11` のロードが真に依存しているのはスレッド 1 の `i10` のストアである．このようなとき，正しく同期ができないため，再びメモリ依存違反が生じる．よって，真の依存のストアと異なるスレッドで実行されるストアは，ストアの実行を同期のトリガとすることの妨げになる．

真の依存ストアを特定するために，ロードが真に依存するストアが実行されるコアを予測する．このコアを予測するために，違反が生じたときのストアが実行されたスレッドとロー

© 2010 Information Processing Society of Japan

ドが実行されたスレッドの間隔を記録して，同期に使用する．すなわち，ロードが実行されるコアから，記録されたスレッド間隔だけ先行するコアのストアのみをトリガとして同期を行う．

もし，現在記録しているスレッド間隔を用いて同期を行い，再び違反が生じたならば，予測に用いるスレッド間隔を更新する．これにより，最終的に真の依存ストアが存在する正しいスレッド間隔で同期ができると考えられる．

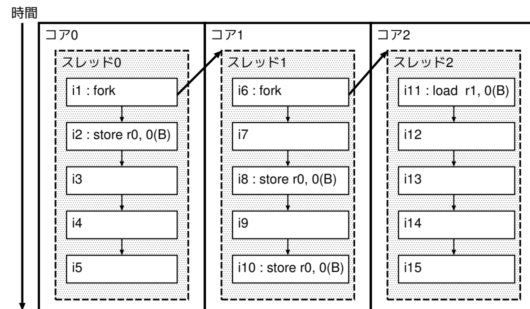

図 2 依存のパターンを示す例

3.4 同一コア内の偽の依存への対処

1 つのスレッドにおいて，複数のストアが同一のデータ・アドレスに書き込みを行うことがある．例えば，図 2 のスレッド 1 においてデータ・アドレス B に対して，2 回ストアが実行されているような場合である．真に依存しているストアとの同期を行うためには，3.3 節で述べたスレッド間隔を用いた同期方式は十分でない．

そこで，ロードと同一のデータ・アドレスに書き込むストアの実行回数を予測することとする．この回数を予測するために，ロードが引き起こした違反の回数を記録して，使用する．ロードのコアからあるスレッド間隔離れたコアにおいて，この回数だけストアが実行されると予測し，最後のストアをトリガとしてロードの同期を行う．これによって，真の依存ストアに対して同期ができる．

4. 提案するメモリ依存予測及び同期機構

本章では提案するメモリ依存予測/同期機構について説明する．まず，構成について説明する．次に，機構の動作を説明する．最後に，メモリ依存に対してどのように動作するかを具体例を挙げて示す．

4.1 メモリ依存予測/同期機構の構成と動作

メモリ依存予測/同期機構の構成について説明する．メモリ依存予測には，**メモリ依存予測表**と呼ぶ表を使用する．同期には，**メモリ同期表**と呼ぶ表と**拡張した LSQ**(Load / Store Queue：ロード・ストア・キュー) を使用する．

4.1.1 メモリ依存予測表

依存するロードとストアの関係 (依存，スレッド間隔，ロードが依存するストアがスレッドにおいて何回目に実行されるか) は実行において変化しないという考えに基づいて，メモリ依存違反を起こした命令の PC とスレッド間隔，違反回数をメモリ依存予測表に記録する．記録された情報を基に依存関係の予測を行う．

メモリ依存予測表には，ストア用とロード用のものを用意する．メモリ依存予測表はすべてのコアで共有される．

4.1.2 ストア用のメモリ依存予測表

ストア用のメモリ依存予測表を図 3(a) に示す．同表の各エントリは，次のフィールドで構成される．

- V：エントリが有効であることを示すフラグ
- PC タグ：依存を持つと予測されるストアの PC タグ

メモリ依存違反が生じたとき，ストア命令の PC をインデクスとして，メモリ依存予測表を参照する．V ビットを有効化し，ストア PC タグを記録する．

4.1.3 ロード用のメモリ依存予測表

ロード用のメモリ依存予測表の構成を図 3(b) に示す．同表の各エントリは次のフィールドで構成される．

- V：エントリが有効であることを示すフラグ
- PC タグ：依存を持つと予測されるロードの PC タグ
- Interval：メモリ依存違反スレッド間隔
- Count：メモリ依存違反回数

メモリ依存違反が生じたとき，ロード命令の PC をインデクスとして，メモリ依存予測表を参照する．V ビットを有効化し，ロード PC タグを記録する．また，違反スレッド間隔 (Interval) を記録して，違反回数 (Count) をカウント・アップする．

Interval は，次の式を用いて導かれる (N はコア数).

$$Interval = (\text{ロードが実行されたコア番号} - \text{ストアが実行されたコア番号} + N) \ mod \ N$$

© 2010 Information Processing Society of Japan

Vol.2010-ARC-188 No.1
2010/3/1

メモリ依存予測表に記録するとき，フィールドに Interval が異なる有効なエントリが存在した場合は，真の依存を予測するために Interval が小さいものを優先して残す．

予測表は一定サイクルごとにリセットする．偽の依存個数は動的に変動すると思われるが，Count には，減少の仕組みを持ち合わせていないためである (上述したように，依存違反が生じるごとにアップするのみである)．

4.1.4 メモリ同期表

メモリ同期表の構成を図 3(c) に示す．同表は，動的なストアと動的ロードを同期させるための情報を登録する．具体的には，ストアとロードを関連付けるために，データ・アドレスを ID として登録する．また，ロードが依存するストアを識別するために，予測表の情報を用いて,「ストアが実行されるコア」,「そのコアにおいて同一データ・アドレスに対して何回ストアが実行されるか」を予測して登録する．

1 つのコアに 1 つのメモリ同期表を用意する．コアがスレッドを終了して，コアが解放されるとき，メモリ同期表はクリアされる．つまり，同期は各スレッドごとに行われる．

メモリ同期表の各エントリは次のフィールドで構成される．

- V : エントリが有効であることを示すフラグ
- データ・アドレス・タグ : 同期を行うロードのデータ・アドレスのタグ
- Core : ロードが依存するストアが実行されると予測したコア番号
- Num : 実行されると予測したストアの数

ロード実行時，ロード用の予測表にヒットした場合に，ロードのデータ・アドレスをインデクスとして同期表が引かれる．同期表のエントリの V ビットを有効化し，データ・アドレス・タグを登録する．また，予測表の情報を用いて，同期用の情報 Core と Num を登録する．そして，ロードの実行をキャンセルして LSQ で待ち合わせる．

Core と Num は，予測表の Interval と Count から次の式を用いて導かれる (N はコア数)．

$Core = (\text{ロードが実行されるコア番号} - Interval + N) \ mod \ N$

$Num = Count$

ストア実行時，ストア用の予測表にヒットした場合に，ストアはデータ・アドレスとストアが実行されたコア番号を後続コアの同期表に対して放送する．同期表においては，放送されてきた情報に対し，データ・アドレス・タグと Core が一致したとき，Num をカウントダウンする．Num が 0 になったとき，同期表エントリのロードが依存するストアが実行されたと判断して，このロードの実行をトリガする．

4.1.5 拡張した LSQ

前述したように，依存が予測されたロード命令は，LSQ で実行を待ち合わせる．拡張した LSQ にロードが依存するストアが実行されると予測したコア番号を記録する．この番号は，依存すると予測したストアが実行されなかったとき，ロード命令の実行はデッドロックされるため，これを解除するのに用いられる．

拡張した LSQ の構成を図 3(d) に示す．各エントリは次のフィールドで構成される．

- 通常の LSQ のフィールド
- Core : 依存するストアが実行されると予測したコア番号

ロード実行時，ロード用の予測表にヒットした場合に，ロードの LSQ の Core フィールドに同期表の Core と同じ値を書き込む．

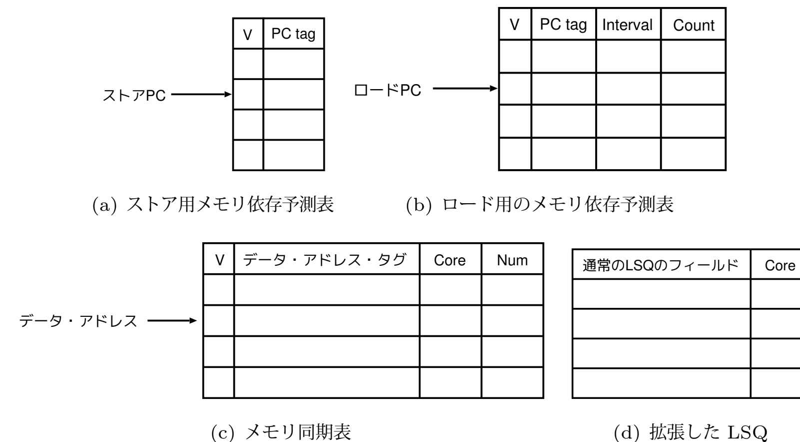

図 3 メモリ依存予測 / 同期機構の構成

4.2 メモリ依存予測/同期機構の動作の説明

提案機構の動作を説明する．提案機構はメモリ依存違反発生時の情報を記録して，この情報を基にメモリ依存関係を予測し，ロードを依存するストアに対して同期させる．

4.2.1 メモリ依存違反の記録の動作

メモリ依存予測表は次の動作によって，メモリ依存を予測するために用いる情報を記録する．メモリ依存違反が起きたとき，メモリ依存違反情報をメモリ依存予測表に記録する．

© 2010 Information Processing Society of Japan

記録する内容は，違反を起こしたストア命令とロード命令の PC，違反が生じたスレッド間隔，違反回数である．

4.2.2 メモリ依存予測の動作

ロード命令を実行する時，ロード用の予測表を用いて，依存を持つかどうかの予測を行う．依存を持つと予測したとき，依存先のストアが実行されたことが通知されるまで，ロードの実行をストールさせる．また，予測表に記録された情報を基に依存関係を予測し，ロードの同期情報を同期表と LSQ に登録する．登録の後，ロードは LSQ で依存すると予測したストアの実行を待ち合わせる．

4.2.3 メモリ同期時の動作

ストア命令を実行するとき，ストア用の予測表を用いて，依存を持つかどうかの予測を行う．依存を持つと予測したとき，ストアのデータ・アドレスとストアが実行されたコア番号を後続コアのメモリ同期表に放送して，ヒットした (データ・アドレス・タグと Core が一致した) 同期表エントリの Num をカウントダウンし，Num が 0 になったエントリのロードを LSQ に通知することによって，ロードの実行をトリガする．

4.2.4 コアにおいてスレッドが終了したときの動作

あるコアにおいてスレッドが終了したとき，終了コアの番号を後続コアに放送して，終了したことを知らせる．これにより，ストアが実行されると予測して実際には実行されないために，実行がデッドロックしているロードの実行をトリガする．

終了したコアの番号を受け取ったコアは，LSQ でストールしているロードの中で，終了コア番号と Core が一致したもののストールを解除する．

4.3 メモリ依存予測/同期機構の動作例

メモリ依存予測/同期機構の動作例を図 4〜7 を用いて示す．4 コア・プロセッサで，4 エントリのメモリ依存予測表 (ストア用，ロード用) と，4 エントリのメモリ同期表を仮定する．

4.3.1 メモリ依存違反の記録の動作例

図 4 に示す `i8` のストアと `i14` のロードで生じたメモリ依存違反が検出されて，違反情報が記録されるときの動作を説明する．

まず，図 5 の上に示すように，`i8` のストアに対応するストア用の予測表のエントリの V ビットを有効化し，PC タグを記録する．

次に，図 5 の下に示すように，`i14` のロードに対応するロード用の予測表のエントリの V ビットを有効化し，PC タグ，Interval : 2，Count : 1 を記録する．Interval の計算は次のように行う．

$$
\begin{aligned}
Interval &= (i14\ \text{のロードが実行されたコア番号} - i8\ \text{のストアが実行されたコア番号} \\
&\qquad + N)\ mod\ N \\
&= (3 - 1 + 4)\ mod\ 4 = 2
\end{aligned}
$$

4.3.2 メモリ依存予測時の動作例

図 6 に示すように，コア 0，1，2，3 においてスレッド 11，12，13，14 が実行されているとする．

スレッド 13 の `i110` のロードが実行されるときを考える．このとき，ロード用の予測表の対応するエントリを参照する (図 5 の下)．予測表ヒット (V ビットが有効，かつ PC タグが一致) したため，ロードは依存を持つと予測する．このとき，ロードのデータ・アドレス B で指される同期表のエントリの V ビットを有効化し，データ・アドレス・タグを登録する．さらに，予測表に記録された情報を基に，依存するストアが実行されるコア番号 (Core = 0)，及びストアの実行回数 (Num = 1) を登録する．Core と Num の計算は次のように行う．

$$
\begin{aligned}
Core &= (i110\ \text{のロードが実行されるコア番号} - Interval + N)\ mod\ N \\
&= (2 - 2 + 4)\ mod\ 4 = 0 \\
Num &= Count = 1
\end{aligned}
$$

ロードはストアが実行されるのを LSQ で待ち合わせる．メモリ同期表の Core = 0 を LSQ の Core フィールドにも同様に登録する (図 7)．

4.3.3 メモリ同期時の動作例

図 6 において，`i104` のストアが実行されるときを考える．このストアに対応するストア用の予測表を参照する (図 5 の上)．予測表ヒット (V ビットが有効，かつ PC タグが一致) したため，ストアは依存を持つと予測する．このとき，ストアのデータ・アドレス B とストアが実行されたコア番号 0 を後続コアの同期表に対して放送する．

放送されてきたデータ・アドレス B とコア番号 0 が，コア 2 の同期表にヒット (データ・アドレス・タグと Core に一致) したため，同期表の Num をカウントダウンする (図 6)．このとき，Num が 0 となるため，ロードが依存するストアが実行されたことがわかる．そこで，データ・アドレス B を LSQ に放送し，LSQ で実行を待ち合わせている `i110` のロードの実行をトリガする (図 7)．

5. 評　　価

本章では，まず評価環境について説明する．次に，提案機構の性能評価を行う．

© 2010 Information Processing Society of Japan

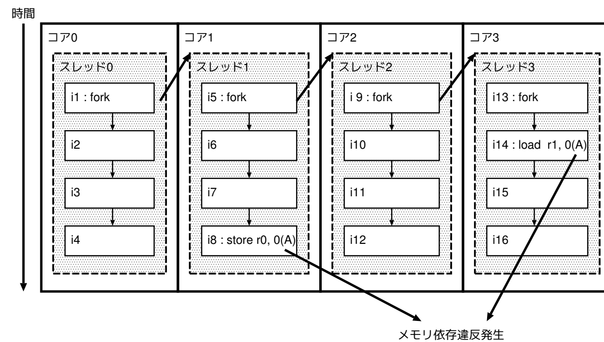

図 4 メモリ依存違反発生の例

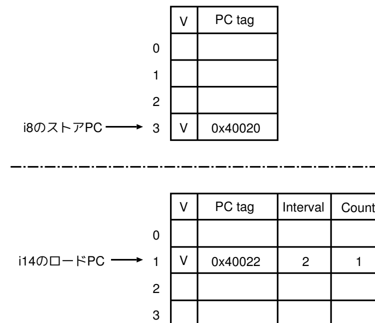

図 5 検出した違反の情報を予測表に書き込む動作例 (上： ストア用，下： ロード用)

5.1 評価環境

ベンチマーク・プログラムとして，SPECint2000 の 4 本，SPECfp2000 の 5 本，Mediabench の 5 本を使用した．これらは SKY 用コンパイラによって並列化されたものである．並列化する前のベンチマークのバイナリは，GNU GCC Version 2.7.2.3(コンパイル・オプション： -O6 -funroll-loops) を用いて作成した．評価はトレース駆動シミュレータを用いて行った．トレースは SimpleScalar Tool Set Version 3.0 を利用して採取した．

表 1 にコアの基本構成を示す．評価の基準となる単一コア・プロセッサは，SKY を構成する 1 つのコアプロセッサとした．表 2 にキャッシュとメモリの基本構成を示す．キャッシュとメモリは全コアで共有されている．評価に用いるコア数は 8 とした．

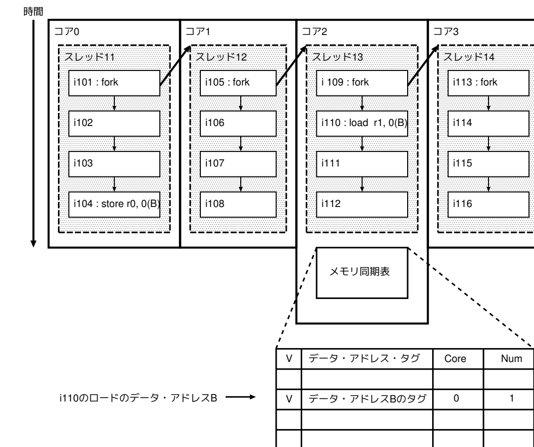

図 6 メモリ依存予測/同期の動作例

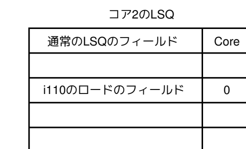

図 7 拡張した LSQ の動作例

表 1 コアプロセッサの構成

Pipeline width	4-instruction wide for each of fetch, decode, issue, commit
Instruction window	64 entries
LSQ	64 entries
ROB	128 entries
Function unit	4 iALU, 1 iMULT/DIV, 4 Ld/St, 4 fpALU, 1 fpMULT/DIV/SQRT, 4 Send
Branch prediction	PAp(4-bit history, 16K-entry PHT), 10-cycle misprediction penalty

5.2 性能評価

提案機構の性能評価を行う．

5.2.1 評価モデル

以下のモデルについて評価した．

ⓒ 2010 Information Processing Society of Japan

表 2 キャッシュとメモリの構成

L1 I-cache	64KB, 2-way, 32B line
L1 D-cache	64KB, 2-way, 32B line, 2-cycle hit latency
L2 cache	2MB, 4-way, 64B line, 12-cycle hit latency
Main memory	300-cycle minimum latency, infinite bandwidth

- 単一スレッド ： 単一コア・プロセッサで，単一スレッドを実行するモデル
- ブラインド ： 予測及び同期を行わずにスレッドを投機実行するモデル
- 提案機構の動作を部分的に制限したモデル
  - ストール ： 予測表に記録されたロードの実行を先行スレッドすべてが終了するまでストールさせるモデル
  - 予測 & ストール ： 予測表に記録されたロードの実行を依存するストアが実行されると予測したコアのスレッドが終了するまでストールさせるモデル
  - 予測 & 1 回同期 ： 予測表に記録されたロードの実行を依存するストアが実行されると予測したコアで，ロードと同一のデータ・アドレスにアクセスする 1 回目のストアに同期させるモデル
- 予測 & カウント同期 ： 提案機構のモデル．予測表に記録されたロードの実行を依存するストアが実行されると予測したコアで，ロードと同一のデータ・アドレスにアクセスするストアをカウントして同期させるモデル
- 完全同期 ： メモリ依存を完全に同期させる理想のモデル

表 3 各モデルにおける提案機構の有効な動作

	ブラインド	ストール	予測 & ストール	予測 & 1 回同期	予測 & カウント同期
ロードを識別してストール	x	o	o	o	o
ストアの実行コアを予測	x	x	o	o	o
同期 (ストア 1 回)	x	x	x	o	o
同期 (ストアをカウント)	x	x	x	x	o

提案機構の動作を部分的に制限した各モデルが，提案方式のどの特徴を有効化あるいは無効化されているかを表 3 にまとめる．

提案機構を使用するモデルでは，メモリ依存予測表 (ストア用，ロード用) は，リセット間隔 1M サイクル，256 エントリとした．また，メモリ同期表は 8 エントリとした．

5.2.2 性 能 比 較

各モデルの IPC を図 8 に示す．また，各モデルの「完全同期」モデルからの性能低下率を図 9 に示す．

「ブラインド」モデルでは ammp，vortex，ammp の IPC が「単一スレッド」モデルを下回っている．これらではメモリ依存違反によるペナルティが非常に大きいことがわかる．逆に，mgrid，mesa，adpcm_dec，mpeg2enc では,「ブラインド」モデルは完全同期モデルに対して IPC が低下していない．メモリ依存違反が発生しないか，違反のペナルティが小さいからである．

5.2.2.1 「ブラインド」に対する提案機構モデルの性能向上率

「予測 & カウント同期」は,「ブラインド」に対して幾何平均で 7.6%の性能向上を示す．これらの性能向上は,「ブラインド」で「完全同期」から性能が非常に低下した parser，vortex，ammp に対して,「予測 & カウント同期」では性能低下を小さくできたために得られた．また，それ以外のベンチマークでも「ブラインド」は大きな性能低下を示しているが,「予測 & カウント同期」は性能低下を小さくできている．

以上のことから，提案機構は依存を予測して同期することにより，違反を回避し性能向上できる．よって，本手法は有効である．

次に，メモリ依存予測/同期機構の各特徴の有効性について，それぞれ述べる．図 9 に示した，各モデルの「完全同期」モデルからの性能低下率で比較する．

5.2.2.2 「ブラインド」と「ストール」モデルの比較

「ブラインド」の「完全同期」からの性能低下率は幾何平均で 9.6%と非常に大きい．これはメモリ依存違反のペナルティが大きいためであると考えられる．

これに対し,「ストール」の性能低下率は幾何平均で 4.6%と,「ブラインド」と比べると非常に小さい.「ブラインド」の性能低下率に対して，4.9 ポイントの性能低下を削減できた．この削減が大きいことから,「ストール」の手法によって，違反を回避して性能低下を削減できることがわかる．よって，依存を持つロード命令を識別して，先行スレッドすべてが終了するまでストールさせることは有効である．ただし，mesa と mpeg2dec は若干性能が低下した．これはストールによる影響が違反によるペナルティを上回ったためと考えられる．

5.2.2.3 「ストール」と「予測 & ストール」モデルの比較

「予測 & ストール」モデルの「完全同期」からの性能低下率は，幾何平均で 3.4%である.「ストール」に対して，1.2 ポイントの性能低下を削減できた．ただし，削減は小さい．

ベンチマークごとに見ると,「予測 & ストール」は parser，ammp，mesa，g721dec,

© 2010 Information Processing Society of Japan

g721enc で性能低下を削減できている．これらではストアの実行コアを予測して，そのコアのスレッドが終了するまでロードの実行をストールさせる効果が表れている．これらでは依存が生じるスレッド間隔が大きいために，実行コアを予測する効果が表れると考えられる．つまり，先行スレッドすべての終了を待つのではなく，ストアが実行されると予測したコアのスレッドだけの終了を待つことによってストール期間を削減できている．

以上のことから，ストアの実行コアの予測は有効であると言える．ただし，依存が生じるスレッド間隔が小さいとき，コアを予測してもストール期間の削減が小さいため，コアの予測効果は小さい．全体として，依存が生じるスレッド間隔は小さい傾向にあると思われるため，全体の性能低下の削減は小さいものとなったと考えられる．

5.2.2.4 「予測 & ストール」と「予測 & 1 回同期」モデルの比較

「予測 & 1 回同期」モデルの「完全同期」からの性能低下率は，幾何平均で 3.3%である．「予測 & ストール」に対して，0.1 ポイントの性能低下を削減できたが，いくつかのベンチマークでは性能が悪化した．

性能が悪化したのは，parser，equake，mpeg2enc においてである．これらでは，1 回のストアをトリガとして同期を行うと，複数回ストアが実行されるときに正しい同期を行うことができないため，違反が生じペナルティを被ったと考えられる．その根拠として，違反発生率を図 10 に示す．これは，fork の実行回数に対して，違反が生じた割合を示す．この図において，parser や equake で,「予測 & カウント同期」では違反がほとんど生じていないが,「予測 & 1 回同期」では違反が生じている．gzip と mpeg2dec でも違反が生じているが，性能低下はない．これらでは違反のペナルティが小さいため性能低下していないと考えられる．

ammp，g721dec，g721enc では性能向上している．これらでは，1 回のストアをトリガとして同期を行うことによって，ロードのストール期間を短縮できたためであると考えられる．

以上のことから，データ・アドレスをトリガとした同期は有効であるが，1 回目のストアをトリガとすることにより，違反が生じ性能低下を引き起こす可能性があることがわかった．

5.2.2.5 「予測 & 1 回同期」と「予測 & カウント同期」モデルの比較

「予測 & カウント同期」モデルの「完全同期」からの性能低下率は，幾何平均で 2.7%である.「予測 & 1 回同期」に対して，0.6 ポイントの性能低下を削減できた．

parser や equake，mpeg2dec では，性能向上が得られた．これらでは，ストアの実行回数をカウントすることにより，正しく同期ができたためであると考えられる．よって，ストアの実行回数を予測して同期することは有効である．

多くのベンチマークでは性能に大きな変化はなかった．図 10 に示すように，これらのベンチマークでは「予測 & 1 回同期」のときの違反の割合が小さいため，1 回のストアによる同期で十分と言える．

5.2.2.6 「予測 & ストール」と「予測 & カウント同期」モデルの比較

「予測 & カウント同期」モデルは,「予測 & ストール」に対して，0.7 ポイントの性能低下を削減できた．

全体としては，性能低下の削減は小さい．これはスレッドが逐次的で，ロードが先行スレッドの終了間際のストアに依存しているためと考えられる．つまり，スレッドの終了までストールするときのストール・サイクルと，同期時の同期サイクルに大きな差がないため，性能の差が小さいと考えられる．

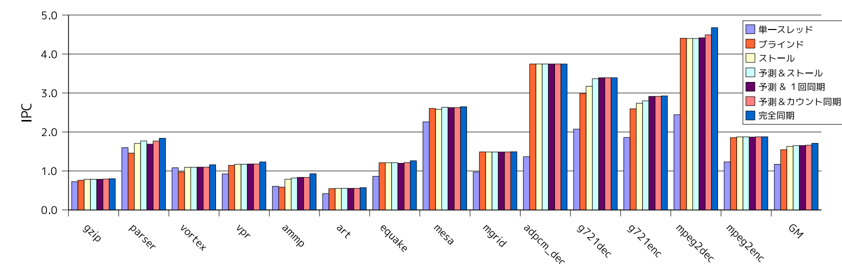

図 8 IPC(8 コア)

6. 関 連 研 究

マルチスレッド実行におけるメモリ依存予測/同期を行う機構の研究がされている[7)–9)]．

Moshovos ら[7)] は，MDPT/MDST(Memory Dependence Prediction Table / Memory Dependence Syncronization Table) を提案している．MDPT/MDST は静的な依存を識別し，静的なストアとロードの組に対応する動的なインスタンスがメモリ依存違反を引き起こすかどうかの予測を行う．しかし，同期においてデータ・アドレスを考慮していないため，誤ったストアとロードの組に依存があると予測する可能性がある．

Cintra ら[8)] は，メモリ・ライン単位のメモリ依存予測を行っている．LDE(Late Disam-

© 2010 Information Processing Society of Japan

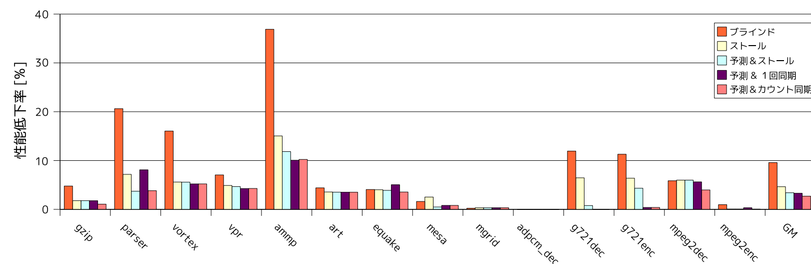

図 9 「完全同期」からの性能低下率

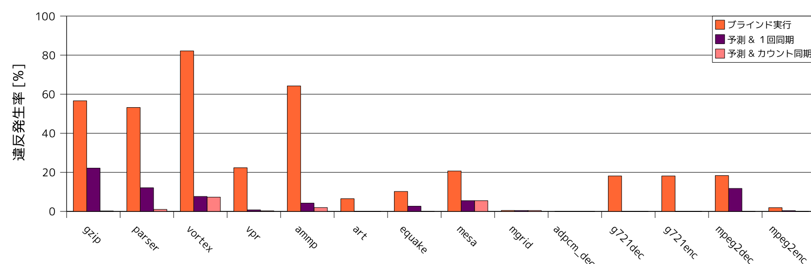

図 10 違反発生率

biguation Engine) により，メモリ・ラインの中のワードに対するメモリ依存違反を検出し，VPT(Violation Prediction Table) にメモリ・ラインにおける違反情報を記録する．依存違反の回数に比例して，依存違反を検出する状態から，値予測を行う状態，ストールして同期を行う状態に遷移することによって，メモリ依存違反を回避する．

Roesner[9] らは，ストアの実行回数を予測し，同期を行う手法を提案している．一度違反を起こしたロード命令は，すべての先行ストアが実行されるまで実行を停止し，その間に実行されたストアの数をカウントし，予測に用いる．

7. ま と め

本論文では，メモリ依存予測及び同期機構を提案した．本機構はメモリ依存を持つロード命令を識別し，ストア命令が実行されるコアと実行回数を予測することにより，依存するストアに同期させる．マルチコア・プロセッサ SKY において，提案機構の評価を行ったところ，提案機構は 8 コア使用時に，ブラインド実行時の性能に対して，幾何平均で 7.6%の性能向上が得られた．

参 考 文 献

1) Sohi, G.S., Breach, S.E. and Vijaykumar, T.N.: Multiscalar Processors, *Proceedings of the 22nd International Symposium on Computer Architecture*, pp.414–425 (1995).
2) 鳥居淳, 近藤真己, 木村真人, 池野晃久, 小長谷明彦, 西直樹：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 年 6 月).
3) Steffan, J.G. and Mowry, T.C.: The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization, *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pp.2–13 (1998).
4) Renau, J., Tuck, J., Liu, W., Ceze, L., Strauss, K. and Torrellas, J.: Tasking with Out-of-Order Spawn in TLS Chip Multiprocessors: Microarchitecture and Compilation, *Proceedings of the 19th International Conference on Supercomputing*, pp. 179–188 (2005).
5) Kobayashi, R., Iwata, M., Ogawa, Y., Ando, H. and Shimada, T.: An On-Chip Multiprocessor Architecture with a Non-Blocking Synchronization Mechanism, *Proceedings of the 25th EUROMICRO Conference*, pp.432–440 (1999).
6) 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・アークテクチャSKY，情報処理学会論文誌， Vol.42, No.2, pp.349–366 (2001 年 2 月).
7) Moshovos, A., Breach, S.E., Vijaykumar, T.N. and Sohi, G.S.: Dynamic Speculation and Synchronization of DataDependences, *Proceedings of the 24th International Symposiumon Computer Architecture*, pp.181–193 (1997).
8) Cintra, M. and Torrellas, J.: Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors, *Proceedings of the 8th International Symposium onHigh-Performance Computer Architecture*, pp.43–54 (2002).
9) Roesner, F., Burger, D. and Keckler, S. W.: Counting Dependence Predictors, *Proceedings of the 35th International Symposium on Computer Architecture*, pp. 215–226 (2008).

© 2010 Information Processing Society of Japan