

## スレッドレベル並列化のためのメモリアリネーミング Memory Renaming for Thread-Level Parallelization

藤井 崇弘<sup>†</sup> 森田 清隆<sup>†</sup> 布目 淳<sup>‡</sup> 平田 博章 柴山 潔<sup>‡</sup>  
Takahiro Fujii Kiyotaka Morita Atsushi Nunome Hiroaki Hirata Kiyoshi Shibayama

### 1. はじめに

マルチスレッドプロセッサやマルチコアプロセッサが商用のマイクロプロセッサとして市販される一方で、逐次実行を前提として記述されたプログラムから自動的にスレッドレベルの並列性を抽出することは依然として困難な状況にある。そこで、プログラム中のできるだけ外側のループを対象として、その各イタレーションをスレッドとして並列に実行するものと想定した場合にどのような要因が並列実行（並列性抽出）の障害となるかを調べた。その結果、スレッド間に依存関係を生じさせる変数の多くは、スレッド内での最初のアクセスが書き込みで始まっていることが判明した[1]。つまり、その変数の使用に関してスレッド間で依存関係が生じるが、変数の値そのものを後続のスレッドで使用するわけではなく、このような本質的でない依存関係のために並列実行の効果が期待できないものとなっている。

命令レベルでこのような依存関係が生じる場合、例えばスーパースカラプロセッサなどでは、レジスタリネーミングによって依存関係を除去している。概念的には同じアプローチでスレッド間の依存関係を除去することができるが、並列処理の粒度の違いから、メモリ上のデータに対してリネーミングを行う方式を開発しなければならない。また、その前提として、メモリ上のデータに対する依存解析を行う必要があり、この点については過去にも試み[2][3]がなされている。

本稿では、キャッシュを用いてメモリ上のデータのアクセスに関するスレッド間の依存解析を行うとともに、メモリ上のデータに対してリネーミングを行う方式を提案する。本稿で提案するメモリアリネーミングのみではすべての種類の依存関係を取り除くことは不可能であり、完全なスレッドレベル並列化を達成するまでには至らないが、並列性抽出の可能性について報告する。

### 2. メモリアリネーミング

#### 2.1 メモリアリネーミングの概要

逐次実行を前提として記述されたプログラムにおいて、できるだけ外側のループを対象に、その各イタレーションをスレッドとして投機的に並列実行する。実行環境としてマルチコアプロセッサを想定し、1個のプロセッサコアで1個のスレッドを実行するものとする。プロセッサコアごとにデータキャッシュを備え、各データキャッシュの容量は十分に大きく、メインメモリへの追い出しが発生しないものと仮定する。

1個の変数に対してそれぞれのキャッシュにその変数の別バージョンを格納することで、メモリアリネーミングを

<sup>†</sup> 京都工芸繊維大学大学院 工学科学研究科 情報工学専攻  
<sup>‡</sup> 京都工芸繊維大学大学院 工学科学研究科 情報工学部門  
Dept. of Information Science, Kyoto Institute of Technology

実現する。スレッド間の依存関係およびメモリアリネーミングの制御を行うために、データキャッシュ内に各変数へのアクセス履歴を表す情報を保存する。具体的には、変数単位で（実際にはアドレスごとに）アクセス履歴を保存するためのV状態ビットを、また、キャッシュのライン単位でアクセス履歴を保存するL状態ビットを設ける。

##### 2.1.1 V状態ビット

変数へのアクセス履歴を保存するために2ビットのV状態ビットを設ける。V状態ビットは、以下に示す2種類のビットから成る。

- VFR ビット  
ある変数に対する最初のアクセスが読み出しであること (First Read) を示す。
- VW ビット  
ある変数に対して書き込みが行われたこと (Written) を示す。

また、キャッシュにアクセスする際に、以下に示す条件でV状態ビットをセットする。

- VFR ビットをセットする条件  
VFR ビットと VW ビットがいずれもセットされていない状態で変数の値を読み出すとき。
- VW ビットをセットする条件  
変数に値を書き込むとき。

##### 2.1.2 L状態ビット

キャッシュのラインへのアクセス履歴を保存するために2ビットのL状態ビットを設ける。L状態ビットは、以下に示す2種類のビットから成る。

- LFR ビット  
初回のアクセスが読み出しである変数がライン中に1個以上存在することを示す。
- LW ビット  
ライン中のすべての変数に対して書き込みが行われたことを示す。

また、キャッシュにアクセスする際に、以下に示す条件でL状態ビットをセットする。

- LFR ビットをセットする条件  
ライン中のすべての変数の VFR ビットと VW ビットがいずれもセットされていない状態で、ライン中の変数の値を読み出すとき。
- LW ビットをセットする条件  
変数に値を書き込むことで、ライン中のすべての変数の VW ビットがセットされ、かつ、すべての VFR ビットがリセットされているとき。

なお、キャッシュミスが発生した場合は、キャッシュに新たなラインを確保し、そのラインのL状態ビットとそのラインに含まれる変数のV状態ビットをリセットする。その後、上記の条件で状態ビットをセットする。

## 2.2 スレッド間の依存解析

複数のスレッドを並列実行するときに、V状態ビットとL状態ビットを用いてメモリアクセスに関するスレッド間の依存解析を行う。説明を簡単化するため、ここでは1ラインに1変数のみが格納されるものとする。このときV状態ビットとL状態ビットは同一視できるため、以下ではV状態ビットとL状態ビットをまとめて状態ビット、VFRビットとLFRビットをまとめてFRビット、VWビットとLWビットをまとめてWビットとして説明する。

### 2.2.1 変数の値を読み出すときの処理

あるスレッド(スレッド番号を $k$ とする)で変数 $A$ の値を読み出すときの処理を図1に示す。ただし、スレッドの番号は最も先行しているスレッドから順に0番から割り当てるものとする。

1. 自スレッドの番号 $k$ を $i$ とし、スレッド $i$ のキャッシュに保存されている変数 $A$ の状態ビットを調べる。
  - FRビットとWビットのどちらか一方でもセットされている場合、手順2へ進む。
  - FRビットとWビットがセットされておらず、 $i$ が0でない場合、手順3へ進む。
  - FRビットとWビットがセットされておらず、 $i$ が0である場合、手順4へ進む。
2. スレッド $i$ のキャッシュから変数 $A$ の値を読み出し、スレッド $k$ のキャッシュに保存されている変数 $A$ の状態ビットを調べる。
  - FRビットとWビットがセットされていない場合、手順5へ進む。
  - FRビットとWビットのどちらか一方でもセットされている場合、終了する。
3.  $i$ の値を1だけ減らし、スレッド $i$ のキャッシュに保存されている変数 $A$ の状態ビットを調べる。
  - FRビットとWビットのどちらか一方でもセットされている場合、手順2へ進む。
  - FRビットとWビットがセットされておらず、 $i$ が0でない場合、手順3へ進む。
  - FRビットとWビットがセットされておらず、 $i$ が0である場合、手順4へ進む。
4. メインメモリから変数 $A$ の値を読み出し、スレッド $k$ のキャッシュに保存されている変数 $A$ の状態ビットを調べる。
  - FRビットとWビットがセットされていない場合、手順5へ進む。
  - FRビットとWビットのどちらか一方でもセットされている場合、終了する。
5. スレッド $k$ のキャッシュに保存されている変数 $A$ にFRビットをセットする。

### 2.2.2 変数に値を書き込むときの処理

あるスレッド $k$ で変数 $A$ の値を書き込むときの処理の流れを図2に示す。ただし、並列実行しているスレッドの数を $n$ とし、それぞれのスレッドの番号を最も先行するスレッドから順に0番から $n-1$ 番とする。また、 $i$ の初期値を $k$ とする。

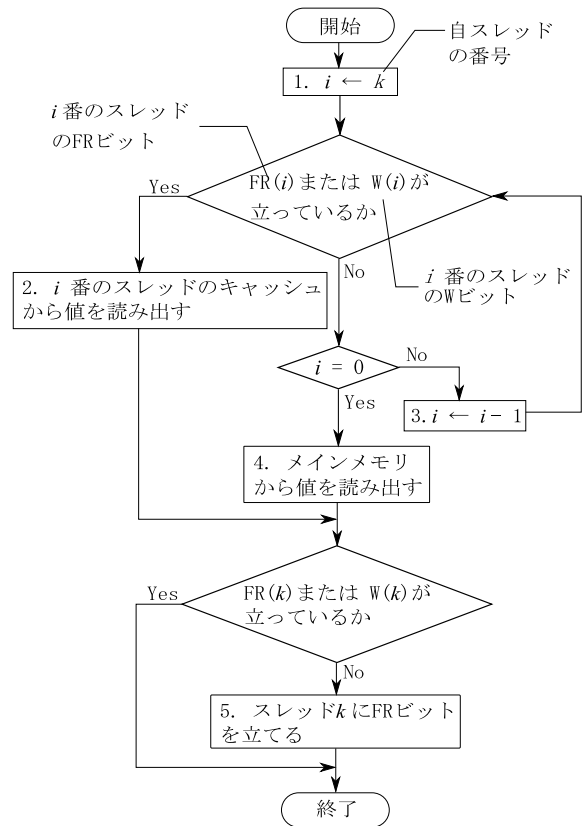


図1 変数を読み出すときの処理

1. 自スレッド $i$ のキャッシュに変数 $A$ の値を書き込み、変数 $A$ のWビットをセットする。
  - 自スレッド $i$ が最後( $i=n-1$ )のスレッドではない場合、手順2へ進む。
  - 自スレッド $i$ が最後( $i=n-1$ )のスレッドである場合、終了する。
2.  $i$ の値を1だけ増やし、スレッド $i$ のキャッシュに保存されている変数 $A$ の状態ビットを調べる。
  - FRビットとWビットがセットされておらず、スレッド $i$ が最後( $i=n-1$ )のスレッドではない場合、手順2へ進む。
  - FRビットがセットされている場合、手順3へ進む。
3. 変数 $A$ に関してスレッド $k$ とスレッド $i$ の間にフロー依存の関係が生じているため、スレッド $i$ を破棄する。
  - スレッド $i$ が最後( $i=n-1$ )のスレッドではない場合、手順2へ進む。
  - スレッド $i$ が最後( $i=n-1$ )のスレッドである場合、終了する。

### 2.2.3 メモリリネーミングの実施例

例として、4個のスレッドを並列実行している状況で、それぞれのスレッドが変数 $A$ へアクセスした場合の処理の様子を図3を用いて説明する。図3において、時刻1以前では変数 $A$ に対するアクセスが発生していないものとする。

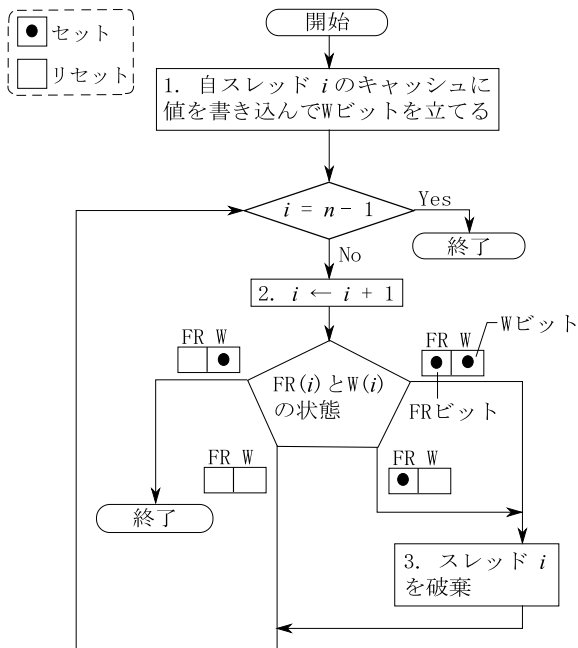


図2 各スレッドで変数を書き込むときの処理

- 時刻1) スレッド3が変数Aに対して読み出しを行うものとする。そのとき、まず、スレッド3のデータキャッシュ上の変数Aの状態ビットを調べる。スレッド3では初めての変数Aへのアクセスであり、FRビットとWビットがセットされていないため、メインメモリから変数Aの値を読み出すとともに、変数AのFRビットをセットする。
- 時刻2) スレッド1が変数Aに対して読み出しを行うものとする。時刻1におけるスレッド3での処理と同様に初めての読み出しアクセスであり、メインメモリから変数Aの値を読み出すとともにFRビットをセットする。
- 時刻3) スレッド2が変数Aに対して書き込みを行うものとする。まず、スレッド2のキャッシュに値を書き込んで、Wビットをセットする。そして、スレッド2の後続スレッドであるスレッド3の状態ビットを調べる。スレッド3のキャッシュで変数AのFRビットがセットされているため、スレッド3を破棄する。
- 時刻4) スレッド0が変数Aに対して書き込みを行うものとする。まず、スレッド0のキャッシュに値を書き込んで、Wビットをセットする。そして、後続であるスレッド1のキャッシュにおいて変数Aの状態ビットを調べる。FRビットがセットされているため、スレッド1を破棄する。また、スレッド2についてはWビットがセットされているため、スレッド2は破棄しない。スレッド2の後続スレッド(スレッド3)については状態ビットは調べない。

文献[3]等の従来の依存解析を行う場合、時刻4でスレッド0が変数Aへ書き込みを行うと、スレッド2との間

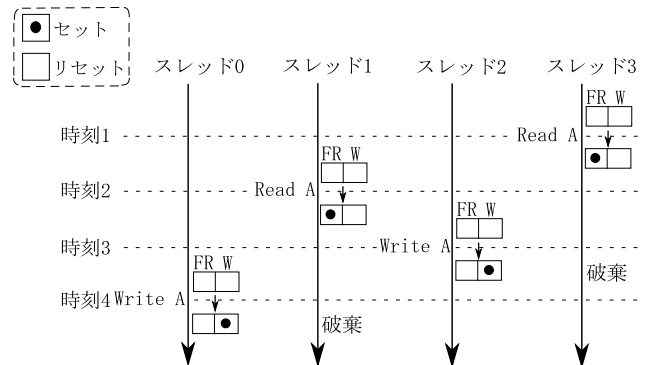


図3 4スレッド実行時の変数アクセスの例

で出力依存によるハザードが生じるため、スレッド2も破棄する必要がある。しかし、メモリリネーミングを行うと、上記のように同時刻4でスレッド2を破棄する必要性を免れ、スレッド2の実行を続行することができる。

### 3. 性能評価

#### 3.1 評価方法

PowerPC[4]の命令セットを対象とした命令レベルのシミュレーションを行い、メモリリネーミングの効果を調べる。評価用のテストプログラムにはSPEC2006ベンチマークプログラム[5]の中の433.milc, 444.namd, 450.soplex, 453.povray, 482.sphinx3(入力データセットはtest)を用いる。それぞれの評価用プログラムの主要部を構成する最外ループを並列化の対象とし、そのイタレーションをスレッドとして投機的に並列実行した場合に、アクセスするメモリ上のデータの依存関係に起因して生じるハザードの発生頻度を見積もる。

#### 3.2 依存関係の除去効果

後続スレッドの投機実行を開始してから、依存関係によるハザードが生じるまでの実行命令数を初期連続実行命令数と呼ぶことにし、そのスレッドの全実行命令数に対する初期連続実行命令数の割合を初期連続実行率と定義する。

1個のラインに1個の変数のみが含まれるものと仮定した場合の初期連続実行率を図4に示す。

メモリリネーミングを行わない場合は、初期連続実行率が高々0.3%程度であり、そのスレッドの実行開始直後に依存関係を破壊するアクセスが生じる。実際の並列実行方式によるが、この時点で先行スレッドとの同期を行うための待ちが生じるか、あるいはスレッドの実行を破棄して再実行を行うことになる。これに対してメモリリネーミングを行った場合は、最大37.2%までスレッドの実行を進めることができ、依存関係による並列実行の阻害要因を大幅に削減できることが分かる。

図4の中には、433.milcと453.povrayのように、メモリリネーミングを行っても依存関係によるハザードが生じるまでの命令数が少数しか増加しないプログラムもある。これは、フロー依存によるハザードが発生する命令が、スレッドの実行開始直後に存在するためである。このよ

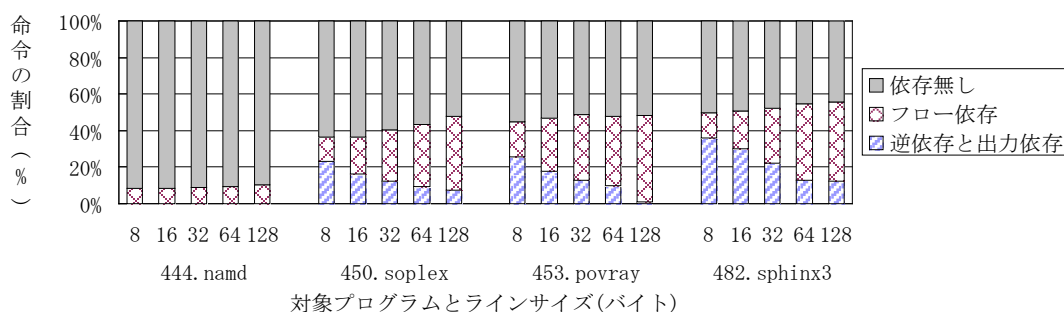


図6 依存関係を生じる命令の割合のラインサイズによる違い

うに、初期連続実行命令数は、フロー依存によるハザードが発生する命令がスレッドのどの部分に存在するかによって異なる。

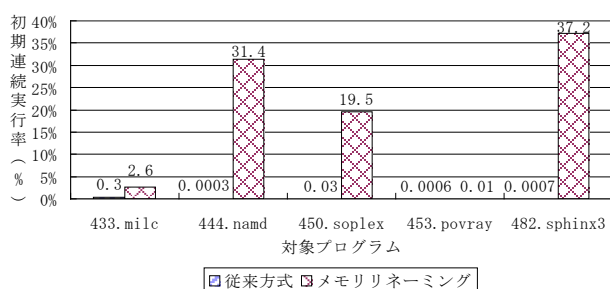


図4 初期連続実行命令数 (変数単位)

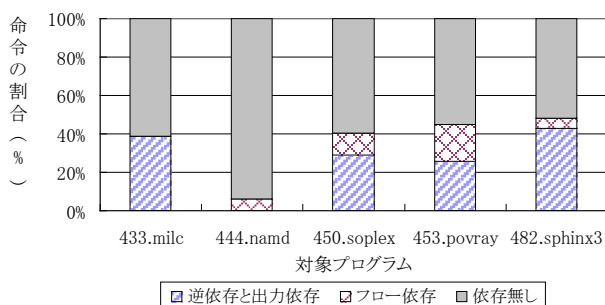


図5 依存関係を生じる命令の割合 (変数単位)

そこで、他のスレッドと依存関係を生じる命令がどれくらい存在するかを計測した結果を図5に示す。図5では、逆依存または出力依存によるハザードが発生する命令と、フロー依存によるハザードが発生する命令、依存関係によるハザードが発生しない命令、の3種に分類し、全実行命令数に占める割合で示した。

メモリアリネーミングを行うことによって、逆依存と出力依存の依存関係を取り除くことができるので、444.namdを除いて、メモリアリネーミングを行うことにより、依存関係によるハザードの発生回数のうち、その半数以上を除去できると考えられる。

### 3.3 ラインサイズによる影響

3.2節では1個のラインに1個の変数のみが含まれるものと仮定したが、これは非現実的である。実際のキャッシュではライン単位で管理し、一般に、1個のライン中に

複数の変数を含む。このような現実的なキャッシュ構成においてメモリアリネーミングの効果を調べる。ラインサイズを8~128バイトの間で変化させた場合に他のスレッドと依存関係を生じる命令がどれくらい存在するかを図6に示す。433.milcにおいては、ラインサイズによる変化が見られなかったため省略する。

1ライン中にフロー依存を生じる変数が存在すると、同じライン中の他の変数が逆依存を生じるものであっても、そのラインはフロー依存を生じるものとして扱う。そのため、図6のようにラインサイズを大きくするとフロー依存によるハザードが発生する命令の割合が大きくなり、逆依存と出力依存によるハザードが発生する命令の割合が小さくなる。メモリアリネーミングを行うことによって、並列性抽出の機会が増大するが、フロー依存をどのように扱うかがより重要な課題になる。

## 4. むすび

本稿では、プログラムのスレッドレベルの並列性抽出の可能性を向上させるためのメモリアリネーミングを提案した。また、シミュレーションによる評価の結果、スレッド間の依存関係によるハザードの発生回数のうち、その半数以上をメモリアリネーミングによって除去できることが分かった。

今後は、フロー依存への対応策とともに有限キャッシュでのメモリアリネーミングの実現方式を検討してゆく。

## 謝辞

本研究の一部は日本学術振興会科学研究費補助金(基盤研究(C)21500053および同22500046)の補助による。

## 参考文献

- [1]森田 清隆, 布目 淳, 平田 博章, 柴山 潔, "スレッドレベル並列化のためのスレッド間依存関係の分類", 第9回情報科学技術フォーラム論文集, vol.1, pp.81-86 (2010).
- [2]Manoj Franklin, Gurindar S. Sohi, "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References," IEEE Transactions on Computers, vol.45, no.5, pp.552-571 (1996).
- [3]Sridhar Gopal, T. N. Vijaykumar, James E. Smith, Gurindar S. Sohi, "Speculative Versioning Cache", Proceedings of the International Symposium on High-Performance Computing, pp.195-205 (1998).
- [4]Freescale Semiconductor, "Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture" (2001).
- [5]Standard Performance Evaluation Corporation, "SPEC CPU2006", <http://www.spec.org/cpu2006/>