1K-8

バイナリコードにおけるスレッド間メモリアクセス依存の実行時チェック手法

豊田 貴志 † 佐藤 智一 †† 大津 金光 † 横田 隆史 † 馬場 敬信 †
† 宇都宮大学工学部情報学科 †† 小山工業高等専門学校 技術室

1 はじめに

バイナリコードを直接マルチスレッド化して並列実行を行う場合に問題となるのは，レジスタ間接指定によるメモリアクセスである．アクセス先のメモリアドレスが不明の場合，どの変数をアクセスしているか判別することができず，スレッド間の依存関係を把握することが困難となる．そこで，マルチスレッド化を行う前にバイナリレベル変数解析を行うことで，レジスタ間接指定のメモリアクセス先アドレスを求め，異なるメモリアクセス同士が同じ変数に対してのアクセスか，異なる変数に対してのアクセスであるかを判別する必要がある．

そこで，バイナリレベル変数解析 [1] を行い，レジスタのデータフローを木構造で表現，正規化することでレジスタのデータフローを比較可能とする．このデータフロー木を用いてメモリアクセス先アドレスを求めることで，異なるメモリアクセス同士が同じ変数に対してのアクセスか，または異なる変数に対してのアクセスなのかを判別する．

しかし，データフロー木を構成する要素に実行時にメモリからロードされるデータが含まれている場合は，異なるデータフロー木でも実行時に同じ値となり依存が発生する可能性が存在する．その場合，マルチスレッド実行を行って正しい結果を得るためには，実行時に依存関係が発生しないことをチェックする必要がある．しかし，マルチスレッド実行中にチェックを行うのは速度向上を妨げる要因となる．そこで，本稿では 3 つのバイナリレベル実行時チェックの手法を提案し，SPECfp95，fp2000 ベンチマークを対象にして実際に実行時チェックを行った際のオーバーヘッドを測定する事によって，それぞれの実行時チェック手法の有効性について評価する．

2 バイナリレベル変数解析

バイナリレベルでマルチスレッド化するためには，対象となるループのイテレーション間依存メモリアクセスを検出する必要がある．バイナリレベル変数解析を行い，メモリアクセスアドレスをデータフロー木で表現することで，木の形を比較してイテレーション間依存メモリアクセスの検出を試みる．しかし，レジスタの値は実行時にメモリからロードされるため，レジスタや定数から構成されるデータフロー木がで表現されるアドレスは実行時でなければ厳密に求めることができない．そのため，データフロー木の形が違う場合でも，実行時に同じ値となりイテレーション間依存メモリアクセスが発生する可能性がある．

バイナリレベルでマルチスレッド化されたコードを実行して正しい結果を得るためには，イテレーション間依存メモリアクセスを実行時にチェックする必要がある．実行時チェックを行うための手法を次節にて述べる．

3 実行時チェック手法

実行時にイテレーション間依存メモリアクセスが発生していないことをチェックするために，まず実行時であれば静的な解析では判別しなかったアドレスも求めることが可能であることに着目して，ループ内の依存が明確ではないロードアドレスとストアアドレスを逐一比較してチェックを行う．この手法を全チェック法と呼ぶことにする．全チェック法は，直接アドレスを比較するので確実なチェックを行うことができる．全チェックによる処理時間の様子を図 1 に示す．図 1(a) 中のロード命令，ストア命令は静的に依存が発生しているかを求めることができなかったとする．この場合，全チェックを適用すると，図 1(b) のようにループ内でストア命令が実行される前にロードアドレスとストアアドレスの比較を行う．そのため処理時間が長くなり，マルチスレッド実行による性能向上を妨げる．

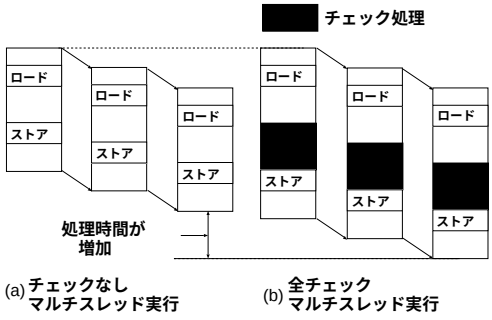

図 1: 全チェックによる処理時間の増加

確実にチェックを行うことができても，チェックのために性能向上を妨げてしまうのではマルチスレッド実行の利点が損なわれる．そこで，低オーバーヘッドな実行時チェック手法としてアドレスの変化の予測性に着目してループに突入する前に，メモリアクセスアドレスの変化範囲を算出することでチェックを行う．この手法をループ事前チェック法と呼ぶことにする．ロード，ストア命令によってアクセスされるメモリアドレスは，実行時でないと特定できない．しかし，静的なバイナリレベル変数解析から，アドレスのデータフロー木を構成する要素が，ループ外で定義されてループ内では更新されないレジスタや，誘導変数レジスタから構成されている場合であれば，アドレスがイテレーションごとに変化するか否か，変化する場合はどのように変化するのかを確実に求めることができる．ループ事前チェック法であれば，全チェック法のようにループ本体のコードを増やすことなく，さらにスレッド毎の比較ではなくループに突入する前にチェックを行うのでチェック回数も抑えることができる．従って，低オー

Binary-Level Runtime Checking Techniques for Resolving Memory Access Dependencies in Multithreaded Codes
Takashi Toyoda†, Tomokazu Satou††, Kanemitsu Ootsu†, Takashi Yokota†and Takanobu Baba†
Department of Information Science, Faculty of Engineering, Utsunomiya University{†}
Technical Office, Oyama National College of Technology {††}

バーヘッドな実行時チェックが実現できる．

アドレスの変化が確定できる場合は，ループ事前チェック法で低オーバーヘッドなチェックが実現できるが，アドレスの変化が確定できない場合にはこの方法は使えない．そのため，変化が確定できない場合も低オーバーヘッドなチェック法が必要である．そこで，全チェックのように全てのロード，ストアアドレスを比較することでチェックを行うのではなく，スレッド内でもループ事前チェック手法のようにアドレスの範囲を求めてチェックを行う．この手法をループ内アドレス範囲チェック法とする．全てのアドレスを逐一比較するよりも，アドレスの範囲を求めて範囲が一致しないことをチェックする方がチェックコストは小さくなると考えられる．ループ内でロード，ストア命令が実行されるたびにそれ以前に実行されたアドレスの範囲を調べ，スレッドの最後でそれぞれのアドレスの範囲を調べる．もし，範囲が一致していなければマルチスレッド実行を継続，範囲が一致した場合はマルチスレッド実行を中断し後続スレッドを破棄した後，回復処理を行いシングルスレッド実行に移行する．

以上の 3 つの実行時チェック手法を実際にアプリケーションに適用して評価を行う．

4 評価

4.1 評価方法

評価にはスレッドパイプライニングモデルシミュレータの SIMCA[2] を使用し，評価対象として．SPECfp95 の *101.tomcatv* の関数 *main*，*175.vpr* からは関数 *try_swap* のホットループを用いた．*175.vpr* は C で記述されているが，*101.tomcatv* は FORTRAN77 で記述されているため，まず f2c FORTRAN to C トランスレータ (version 19940927) を用いて C のソースコードに変換する．その後，SIMCA 用 gcc クロスコンパイラ (version 2.7.2.3) に *101.tomcatv* は最適化オプション-O2，*175.vpr* は-O3 を適用してコンパイルを行い，バイナリコードを生成する．最適化オプション-O2 と-O3 の違いはインライン展開を適用するか否かである．*101.tomcatv* は関数 *main* しかないためインライン展開する必要が無かったので，-O2 を適用した．生成されたバイナリコードに対してマルチスレッド化を適用し，さらに，生成されたマルチスレッドコードに実行時チェック手法を適用する．

評価方法は，まず対象ループの開始から終了までの実行サイクル数を計測し，そのサイクル数を用いて，実行時チェックにかかったコストを以下の式から算出する．実行時チェック手法を適用したマルチスレッドコードのサイクル数をサイクル数 A，チェックを行わない場合のサイクル数をサイクル数 B とする．

$$\text{オーバーヘッド } [\%] = \frac{\text{サイクル数 } A - \text{サイクル数 } B}{\text{サイクル数 } B} \times 100$$

入力データセットに test を用い，スレッドユニット台数を 4 台，8 台，16 台としてシミュレーションを行った．

4.2 評価結果

101.tomcatv のループにあったメモリアクセスアドレスは，イテレーション毎に固定の場合とイテレーションごとに変化する場合があった。変化する場合は，ロード，ストアアドレスともに一定の同じ値ずつ変化するため，ループ事前チェック法を適用することができた．

175.vpr の関数 *try_swap* のループ中のメモリアクセス命令は，全てのストアドレスの値がループ内でロードされて決定されていた．そのため，ストアアドレスの範囲をループ前で調査することができず，ループ事前チェック法を適用することができないため，全チェック法とループ内アドレス範囲チェック法を適用した．

表 1 にチェックにかかるオーバーヘッドを示す．表中では全チェック法をチェック法 A，ループ事前チェック法をチェック法 B とし，チェック法 C をループ内アドレス範囲チェック法とする．

表 1: チェックにかかるオーバーヘッド

チェックオーバーヘッド：*101.tomcatv*			
ユニット台数	4 台	8 台	16 台
チェック手法 A	274.6 %	242.9 %	205.3 %
チェック手法 B	0.819 %	1.476 %	2.583 %

チェックオーバーヘッド：*175.vpr*			
ユニット台数	4 台	8 台	16 台
チェック手法 A	88.42 %	87.80 %	87.81 %
チェック手法 C	165.56 %	132.5 %	108.3 %

101.tomcatv では，全チェック法とループ事前チェック法でオーバーヘッドに大きな違いが出た．全チェックのオーバーヘッドが大きくなったのは，元々のループのコード数が多くメモリアクセス命令が多いため，チェックのためのコード数が増えたことが大きく影響していると考えられる．

175.vpr では全チェック法，ループ内アドレス範囲チェック法両方ともにオーバーヘッドが大きくなった．特に，ループ内アドレス範囲チェック法のオーバーヘッドが全チェック法のオーバーヘッドより大きくなった．これは，チェックを適用した際の増加した命令数がループ内アドレス範囲チェック法の方が多くなってしまったためであると考えられる．

5 おわりに

本稿では，スレッド間依存メモリアクセスのためのバイナリレベル実行時チェック手法を提案した [3]．その結果，マルチスレッド実行中に依存が発生していないことを逐一チェックする手法よりも，アドレスの予測が可能な場合に低オーバーヘッドでマルチスレッド実行を行うことができた．

今後の課題として，チェックオーバーヘッドが大きくなったループ内アドレス範囲チェック手法の改善が挙げられる．

謝辞 本研究は，一部日本学術振興会科学研究費補助金（基盤研究 (B)18300014，同 (C)16500023，若手研究 (B)17700047）および宇都宮大学重点推進研究プロジェクトの援助による．

参考文献

[1] Tomokazu Satou，Kanemitsu Ootsu，Atsushi Tsukikawa，Takashi Yokota，Takanobu Baba，“A Methodology of Binary-Level Variable Analysis for Multithreading，” Proc. 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)，pp.784–789，Nov. 9-11，2004．

[2] J. Huang，“The SImulator for Multithreaded Computer Architecture (Release 1.2)，” http://www.cs.umn.edu/Research/Agassiz/Tools/SIMCA/simca.html.

[3] 豊田 貴志，佐藤 智一，大津 金光，横田 隆史，馬場 敬信，“ バイナリレベル変数解析のための効率的な実行時チェックコードの検討 ”情報処理学会 第 67 回全国大会講演論文集, 講演番号 5ZB-4, pp.1-193~ 1-194, 2005 年 3 月.