5ZB-4

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

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工学科‡

1　はじめに

我々は、シングルスレッドコードをバイナリレベルでマルチスレッド化するシステムを研究している[1]。

バイナリコードでは、元のソースコードと比較してマルチスレッド化に有用な多くの情報を失っている。特に重要なものとして、レジスタによって間接的に指定されるメモリアドレスの情報がある。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事が出来ず、スレッド間の依存関係を把握する事が困難となる。そこで、バイナリレベル変数解析[2]を行い、同一の変数をアクセスしているかを把握する必要がある。

しかし、配列アクセスやポインタによる間接アクセスのように実行時でないとアクセス先のアドレスが分からない場合がある。その場合は、メモリ上の変数等、スレッド間で依存があるデータがこれらのアクセスによって更新されるのかを実行時にチェックする必要がある。

本稿では低オーバーヘッドな実行時チェックの手法を検討し、浮動少数点演算系アプリケーション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、いかに効率良く実行時チェックを行うことが出来るかを評価する。

2　バイナリレベル変数解析

バイナリコードでは、メモリ上に配置された殆どの変数へのアクセスはレジスタを用いた間接指定でのアクセスとなっている。レジスタでのアクセスは再利用されるため同一のレジスタでも異なる値を持つ事があり、レジスタ間接でメモリアクセスを行う場合には異なるメモリを指し示す場合がある。逆に、他のレジスタに格納されている値が同一の値を持ち、同一のメモリを指し示す場合もある。

そこでまず、対象コードをマルチスレッド化する際に必要な情報を得るために、静的なバイナリレベル変数解析を行う。レジスタの内容を解析するために、手続き開始点より関係するレジスタと定数から構成されるデータフロー木を構築する。また、同一アドレスのメモリへのアクセスを検出するために、ロード/ストア命令の際に、アドレス指定に用いられているレジスタとオフセット値を加算したものについてもデータフロー木を構築する。データフロー木の正規化をおよび比較を行い、同一の内容を持つデータフロー木であった場合にアクセスされているメモリは同一であると判断する。さらに、メモリを介して処理されるデータに対しては、仮想レジスタを導入することによって、メモリへの操作を仮想レジスタへの操作とみなすことでレジスタと同様に解析を行う。

しかし、実行前にはアクセス先のメモリアドレスが分からない場合があるので、変数に関する情報を全て静的に解析する事は困難である。例えば、メモリへの配列アクセスの範囲が変数のアドレスと重なった場合、変数の値が更新される可能性がある。スレッド間で依存がある変数の値が配列アクセスにより更新された場合、スレッド間で正しいデータのやりとりを行うことが出来なく、依存関係を守ることが出来なくなりマルチスレッド実行しても正しい結果を得ることはできない。もし、正しい値を得ることができたとしても、偶然そうなっただけかも知れないので、実行時に変数の値が更新されたかどうかのチェックを行う必要がある。

3　実行時チェック

実行時チェック手法として、以下の2つを検討する。

3.1　手法1

手法1は、マルチスレッド実行中にチェックを行う手法である。ストア命令の度に逐一チェックを行うので、かなりのオーバーヘッドが予測されるが、確実にチェックを行うことができる。以下に、手順を説明する。

(1) マルチスレッド実行中にストア命令がある度に、ストア先のアドレスとスレッド間で依存がある変数のアドレスの比較を行う。
(2) 一致したら変数の値が更新されるということなので、マルチスレッド実行を中断しシングルスレッド実行を開始する。一致しない場合は変数の値は更新されないので、そのままマルチスレッド実行を継続する。

3.2　手法2

手法2は、マルチスレッド実行前にストア先のアドレスの範囲を計算する手法である。現段階では、配列アクセスによってストア先が判らない場合に限定してアドレスを計算している。そのため、完全にマルチスレッド化の正当性を示すことはできてない。以下に、手順を説明する。

(1) ストア先のアドレスは、イテレーション毎に変わる。そこで、ストア先のアドレスを示す式を作成する。
(2) (1)で作成した式の初期値S0と変数のアドレスAとの大小を比較する。変数が複数ある場合は、変数が単調増加(単調減少)していく時は、変数のアドレスが1番大きい(小さい)ものと比較を行う。
(3) S0 > A(S0 < A)で変数が単調増加(単調減少)していく場合は、ストア先のアドレスと、変数のアドレスが一致する事はなく、変数の値が更新されることは無いのでマルチスレッド実行を開始する。逆に、S0 < A(S0 > A)で変数が単調増加(単調減少)していく場合は、(1)で作成した式が最終的に得る値Sを算出して、Sの値と変数のアドレとの比較を行う。S > A(S < A)の場合はストア先のア

A Consideration of an Efficient Run-time Check Code for Binary Level Variable Analysis
† Takashi Toyoda、Tomokazu Satou、Kanemitsu Ootsu、Takashi Yokota and Takanobu Baba
‡ Department of Information Science、Faculty of Engineering、Utsunomiya University

ドレスと変数のアドレスが一致する可能性があるので、シングルスレッド実行を開始する。変数が複数ある場合は、変数が単調増加(単調減少)していく時は、変数のアドレスが1番小さい(大きい)ものと比較を行う。

4 評価

実際のバイナリレベルでのマルチスレッド化が困難なアプリケーションとして、SPECfp95の101.tomcatvを対象としバイナリレベル変数解析、マルチスレッド化を行った。対象ループは実行頻度が高く、演算処理を行っており速度向上比が大きい4つの最内ループ(#1〜#4)とした。ただ、これらのループは、実行時チェックを行わなくても正しい結果を得る事ができたが、実行時チェックを行っていないためにマルチスレッド化の正当性は示せていない。そのため、実行時チェックを行い、マルチスレッド実行の正当性を示す必要がある。

評価には、スレッドパイプライニングモデルを実現するアーキテクチャシミュレータSIMCA[3]を用いた。対象バイナリ生成にはf2c Fortran to Cトランスレータと SIMCA用gccクロスコンパイラ(最適化オプション-O2)を用いた。また、ループ(#1〜#4)は三重ループの最内ループなので、最内ループ、最内ループの1つ外側のループ、最外ループそれぞれでバイナリレベル変数解析を行い、実行時チェック対象の変数を検出した。それぞれのループでチェックが必要な変数の数は、ループ#1が6個、ループ#2、#3、#4が5個である。また、それぞれのループ中の総命令数に対するストア命令数の割合を(ストア命令数／総命令数)で示す。ループ#1は(4／299)、ループ#2は(3／70)、ループ#3は(2／57)、ループ#4は(2／40)である。

手法1、2ともにマルチスレッドコード中に、直接アセンブリコードを書くことによって実装した。評価としてチェックコードを挟んでいない場合(図中ではMT)と、手法1のチェックコードを挟んだ場合(図中ではMT+check1)と、手法2のチェックコードを挟んだ場合(図中ではMT+check2)でそれぞれ並列実行可能サイクル数4、8、16でマルチスレッド実行を行いシングルスレッド実行に対する速度向上比を求める。速度向上比を図1に示す。

手法1では、ループ#1の場合全てのユニット台数で速度向上比が劣化した。ループ#2のでは4、8台の場合、ループ#3、#4では4台の場合に、速度向上比が劣化した。しかし、ループ#2、#3で、ユニット台数16の場合は、速度向上比に大きな劣化は見られなかった。以上から、手法1ではストア命令の数が多い程、マルチスレッド実行に対する実行時チェックのオーバーヘッドが大きくなり、速度向上比を大きくを劣化させると考えられる。ただ、ループ#4に関しては、ユニット台数が8、16台の時、2.54倍から2.60倍と若干ではあるが速度が向上した。これは、チェックコードを挟んだことによりキャッシュのヒット率が上がった為であると考えられる。

手法2では、全ての場合で殆ど性能を劣化させることなく実行時チェックを行うことが出来た。例えばユニット台数16台、ループ#1の場合、速度向上比の劣化は9.80倍から9.78倍への変化程度に抑えることが出来た。他の場合も同じ程度に劣化を抑えられている。

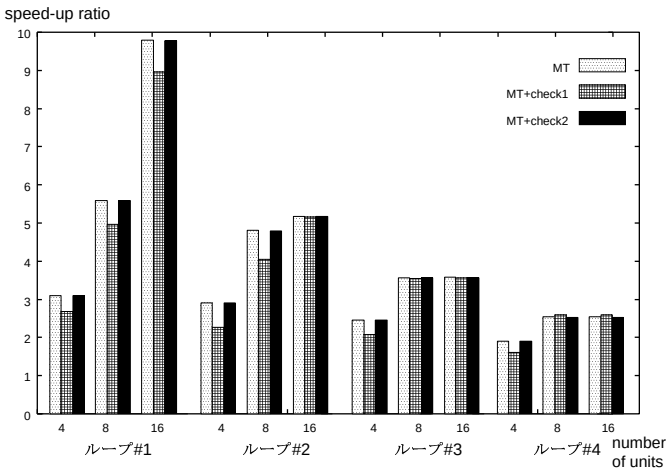

図 1. 速度向上比

5 おわりに

我々の研究で検討されたバイナリレベル変数解析を用いる事により、従来バイナリレベルでのマルチスレッド化が出来なかったループのマルチスレッド化を行うことが出来た。しかし、スレッド間で依存がある変数がメモリ上に置かれているために、これらの変数がロード/ストア命令などのメモリアクセスによって更新されるかどうかのチェックを実行時に行わなければ、正しい結果を得られてもマルチスレッド化の正当性を示すことができなかった。

本稿で用いた手法1を用いることによって、マルチスレッド化の正当性を示すことができた。しかし、手法1では実行時チェックのオーバーヘッドが大きくなり、速度向上比を劣化させてしまう場合がある。

手法2では、スレッド間で依存がある変数が配列アクセスによって更新されないことを低オーバーヘッドで示すことができた。

今後の課題として、手法2で完全にマルチスレッド化の正当性を示すために、ポインタによる間接アクセス等にも対応している、さらなる実行時チェック手法を考案する必要がある。

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)14380135、同(C)16500023、若手研究14780186)の援助による。

参考文献

[1] 大津 金光、小野 喬史、横田 隆史、馬場 敬信、“バイナリレベルマルチスレッド化コード生成手法とその評価、” 情報処理学会論文誌ハイパフォーマンスコンピューティングシステム、Vol.44、 No.SIG-1 (HPS 6)、 pp.70-80、 2003.

[2] 佐藤 智一、月川 淳、大津 金光、横田 隆史、馬場 敬信、“ マルチスレッド化のためのバイナリレベル変数解析手法、”情報処理学会 研究報告、Vol.2004、No.48、 pp.1-6、 2004.05 計算機アーキテクチャ研究会(2004-ARC-158)

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