バイナリレベルポインタ解析を用いた自動マルチスレッド化

佐藤 智一　三木 大輔　横田 昌之　月川 淳　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

アプリケーションのソースコードが参照不可能な場合におけるマルチスレッド化を実現するために、我々はバイナリレベルでシングルスレッドコードからマルチスレッドコードを生成し、実行性能の向上を図るシステム[1]を提案している。

しかし、バイナリは元のソースと比較して、実行性能の高いマルチスレッドコードを生成する際に有用である情報の多くを失っている。その中でも重要なものとして、アクセス先のメモリアドレスについての情報が挙げられる。アクセス先アドレスが不明な場合、ループのイテレーション間での依存を把握することが困難となり、効率の良いコードを生成できない。また状況によってはマルチスレッド化自体が困難となる。

この問題を解決するために、本稿ではまずバイナリレベルポインタ解析の手法を提案する。次に浮動小数点演算系アプリケーションを対象として、提案手法を用いてバイナリレベルでポインタ解析・マルチスレッド化を行い、有効性を検証する。

2　マルチスレッド実行モデル

本研究では、プログラムのループ構造の各イテレーションを、スレッドパイプライニングモデル[2]の概念に基づいてマルチスレッド化を実現する。

スレッドパイプライニングモデルでは、1スレッドをContinuation、TSAG(Target Store Address Generation)、Computation、Writebackの4つのステージに分け、スレッド間データ依存がある場合に同期をとる機能を持ったメモリバッファを用いて依存関係を解決する。Continuationステージでは、誘導変数などのループのイテレーションの実行を開始するのに必要な変数の値の算出を行い、後続スレッドに受け渡す。TSAGステージでは、スレッド間でデータ依存があるメモリのアドレスをメモリバッファに登録する。Computationステージでは、計算処理本体を実行する。このステージでのメモリアクセスは全てメモリバッファに対して行われる。スレッド間依存データについては、専用のストア命令を用いてスレッド間データ通信を行う。Writebackステージでは、スレッドが終了する前にメモリバッファの内容をメモリに書き込む作業が行われる。スレッド間で同期をとり、先行スレッドの書き込みが終了してから書き込みを行う。

3　バイナリレベルポインタ解析

ポインタ解析を行う目的は、レジスタ間接指定のメモリアクセス先のアドレスを求めることである。ポインタ解析の手法と解析情報の利用について説明する。

3.1　ポインタ解析手順

バイナリレベルポインタ解析は以下の手順で行う。

(1) レジスタへのインスタンス番号付加
同名で値の異なるレジスタを区別するために、解析開始点よりレジスタにインスタンス番号を付加する。番号の初期値は#0とし、レジスタの値が定義される毎に番号をインクリメントする。図1に例を示す。

(2) データフロー木の構築
レジスタの内容を解析するために、関係するレジスタと定数から構成される木構造を生成する。図1のレジスタ$2#2について生成した木の例を図2に示す。

(3) ループイテレーション間依存レジスタの検出
ループイテレーション間で依存関係があるレジスタはイテレーション毎に値が変わるため、解析時に定数とみなして扱うことは不可能である。またループ変数として用いられているものがあり、この情報はマルチスレッド化する際に必要となる。

(4) データフロー木の正規化
内容の比較を可能にするため、計算結果が同じ内容を表す木は、一定の構造となるように正規化処理を行う。同じ親を持つ子は一定の順序関係に従って並

```
addiu $29#1, $29#0, -32
sw    $30#0, 24($29#1)
addu  $30#1, $0#0, $29#1
addu  $7#1,  $0#0, $4#0
sll   $2#1,  $7#1, 0x2
addiu $2#2,  $2#1, 14
sll   $2#3,  $2#2, 0x3
sw    $31#0, 28($29#1)
```

図 1. インスタンス番号付加後のコード例

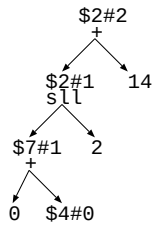

図 2. データフロー木の例

Automatic Binary-Level Multithreading with Pointer Analysis
† Tomokazu Satou, Daisuke Mitsugi, Masayuki Yokota, Atsushi Tsukikawa, Kanemitsu Ootsu, Takashi Yokota, Takanobu Baba
‡ Department of Information Science, Faculty of Engineering, Utsunomiya University

び、簡約可能な演算は存在せず、式で表現した場合の同類項にあたる部分は存在しない状態とする。

(5) 仮想レジスタ割り当て

対象アドレスが固定であるメモリアクセスには、仮想レジスタを割り当てる。メモリへのロード・ストアをレジスタへのアクセスとみなして再び (1) からの処理を行うことで、メモリに格納された値を解析することが可能となり、より深い解析ができる。

3.2 ポインタ解析情報の活用

ループイテレーション間で依存があると検出されたレジスタ (仮想を含む) のうち、1イテレーション内で変化する値が一定であるものを、ループ変数として扱う。

メモリ上にループ変数が置かれている場合はバイナリレベルでの検出が困難であり、従来の方式ではメモリ上のループ変数を検出できないため、マルチスレッド化が不可能であった。しかし、メモリ上に置かれている変数を仮想レジスタとみなして解析することでメモリ上のループ変数を検出することが可能となり、従来バイナリレベルでマルチスレッド化できなかったこの種のループをマルチスレッド化することが可能となる。

またループ変数の変化をもとにして、配列の操作などの、ループのイテレーション間で依存関係があるメモリアクセスを検出できる。

4 評価

実際のマルチスレッド化困難なアプリケーションとして、SPECfp95 の tomcatv を対象とし、ポインタ解析・マルチスレッド化を行って速度向上率を計測した。対象ループは、実行頻度が高く、演算処理を行っている 6 つのループ (#1〜#6) とした。

評価には、スレッドパイプライニングモデル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。

tomcatv のバイナリは、メインルーチンが大きいことと、f2c の変換による影響によって、ほとんどのループ変数がメモリ上に置かれている。そのため従来手法ではループ変数の場所と挙動が不明であり、いずれのループもバイナリレベルマルチスレッド化を行うことが不可能であった。そこで本手法によるバイナリレベルポインタ解析を行い、ループ変数の場所と増減を求めた。解析の結果、対象としたいずれのループについても、ループ変数に同じアドレスのメモリを用いていることがわかった。また、増減値はすべて+1であった。この情報を用いて実際にマルチスレッド化を行い、並列実行可能スレッド数16で速度向上率を計測した結果

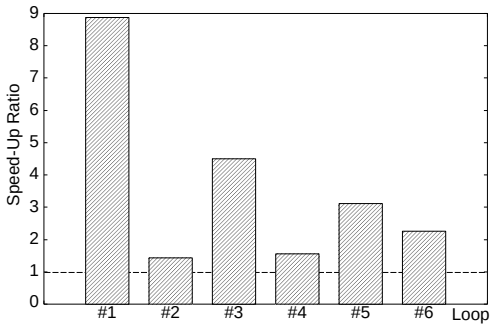

図 3. tomcatv での各ループの速度向上率

を図3に示す。それぞれ逐次実行時と比較して 8.9 倍、1.4 倍、4.5 倍、1.6 倍、3.1 倍、2.3 倍の速度向上を達成している。従来の方法ではマルチスレッド化できなかった tomcatv の各ループについて、ポインタ解析情報を活用することでループ変数を検出し、バイナリレベルマルチスレッド化を実現した。

5 おわりに

本稿では、バイナリレベルで自動マルチスレッド化を行う際に必要となるバイナリレベルでのポインタ解析の手順を示し、実際のアプリケーションを解析することで、従来バイナリレベルでのマルチスレッド化ができなかったループのマルチスレッド化を実現した。また、マルチスレッド化されたアプリケーションの実行速度が逐次実行時と比較して向上することを示した。

今後の課題として、より多くのアプリケーションでの評価によって有効性の検証をすることが挙げられる。また、ループのイテレーション間で依存関係があるメモリアクセスをポインタ解析を利用して検出し、メモリ上でデータ依存がある場合における、より効率の良いマルチスレッドコード生成の実現が挙げられる。

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)14380135、同 (C)14580362、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(HPS6), pp.70-80, 2003.

[2] Jenn-Yuan Tsai, Pen-Chung Yew, “The Superthreaded Architecture: Thread Pipelining with Run-time Data Dependence Checking and Control Speculation,” PACT'96, pp. 35-46, 1996.

[3] Jian Huang, “The SImulator for Multithreaded Computer Architecture (SIMCA), Release 1.2,” http://www-mount.ee.umn.edu/~lilja/SIMCA/index.html.