社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－ARC－158　(1)
2004／5／14

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

佐藤 智一[†]　月川 淳[†]　大津 金光[†]
横田 隆史[†]　馬場 敬信[†]

我々は逐次実行を前提としたプログラムをマルチスレッドコードへバイナリレベルで変換するシステムを研究・開発してきた。効率の良いマルチスレッドコードを生成するためには、スレッド間の依存関係による不要な制約を極力取り除くことが肝要である。しかし、バイナリコードを対象とする場合は、レジスタ間接指定によるメモリアクセス先のアドレスを同定することが困難なため、依存関係の把握自体が困難である。そこで本稿では、まずバイナリレベル変数解析の手順を示す。次に浮動小数点演算系アプリケーションを対象としてバイナリレベルで変数解析を行い、得られた情報を元にバイナリレベルでマルチスレッドコードを生成して有効性を検証する。

A Methodology of Binary-Level Variable Analysis for Multithreading

Tomokazu SATOU,[†] Atsushi TSUKIKAWA,[†]
Kanemitsu OOTSU,[†] Takashi YOKOTA[†] and Takanobu BABA[†]

Currently, we are developing a binary translation system which translates a single-thread binary code into a multithreaded one. In order to generate efficient multithread codes, it is important to remove the unnecessary restrictions by dependencies between threads. However, the binary-level, register-indirect addressing makes it difficult to identify memory access addresses and, therefore, memory access dependencies. In this paper, at first we propose a binary-level variable analysis method. Then, we apply our method to a binary-level floating point application code. Using the variable analysis results, we generate a multithread code and evaluate the effectiveness of the method by the simulation.

1. はじめに

従来、計算機は逐次実行を前提としてその性能を向上させてきたが、信号の伝搬速度や集積度の限界等の物理的な限界によって、高速化には限界が見えてきた。この問題に対してマルチスレッド実行モデルが有効な解決法となるが、マルチスレッド実行を前提としたプログラミングを行うことは容易ではないため、シングルスレッド実行を前提としたコードをもとにしてマルチスレッドコードを自動生成するシステムが必要になると考えられる。しかしながら、アプリケーションのソースコードが参照できない場合は、種々の自動マルチスレッド化コンパイラを用いてマルチスレッドコードを生成することが不可能である。そこで我々は、シングルスレッド実行を前提としたバイナリコードからバイナリレベルでマルチスレッドコードを生成し、アプリケーションの実行性能向上を図るシステムを提案している[1)]。

バイナリコードは元のソースコードと比較して、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、レジスタによって間接的に指定されるメモリアドレスについての情報が挙げられる。アクセス先のメモリアドレスが不明な場合、どの変数をアクセスしているのかを判別することができない。この場合、スレッド間での依存関係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、正しい結果を得るために必要な依存関係を把握することができず、保守的に依存関係を守らせた結果マルチスレッド化自体が困難となる場合がある。

この問題を解決するためには、バイナリレベルで変数解析を行い、どこで同一の変数をアクセスしているかを把握することが必要となる[2)]。そこで本稿ではまず、バイナリレベル変数解析の手順を提案する。次に浮動小数点演算系アプリケーションを対象として、提案手順を用いてバイナリレベルで変数解析を行う。変数解析によって得られた情報を元にバイナリレベルでマルチスレッド化を行って実行性能の向上を図り、バ

† 宇都宮大学工学部情報工学科
Department of Information Science, Faculty of Engineering, Utsunomiya University

イナリレベル変数解析の有効性を検証する。

2. マルチスレッド実行モデル

本研究では、プログラムの実行時間中で大きな割合を占めているループ構造に着目する。ループ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[3]の概念に基づいて各スレッドに割り当てることでマルチスレッド化する。

スレッドパイプライニングモデルでは、1つのスレッドで実行する命令コードをContinuation、TSAG(Target Store Address Generation)、Computation、Writebackの4つのステージに分ける。このスレッドを図1に示すように並列に実行することでマルチスレッド実行を実現する。スレッド間でデータ依存がある場合には、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を実行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。Continuationステージ終了後に後続スレッドを起動し、マルチスレッド実行を開始する。

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。先行しているスレッドがTSAGステージの実行を完了した後、後続のスレッドはTSAGステージの実行に入ることができる。

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ことで正しい値を受け渡すことができる。

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。先行するスレッドよりも先にメモリへの書き込みを行ってしまうと、正しくない結果が保存される可能性がある。そのため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. バイナリレベル変数解析

ループの各イテレーションを単位としてマルチスレッド化する場合、各スレッド間でデータ依存があると、先行スレッドでの変数の操作が終了するまで後続スレッドは実行を中断しなければならない。もし実行を強行してしまえば、スレッド間で値が受け渡されず、正しい実行結果を得られない。また、依存関係が把握できずに依存がない値を待つことになれば、停止しなくて良い場所で実行が停止してしまい、速度向上の妨げとなる。しかしながらバイナリレベルにおいては、変数解析をしなければ変数の所在さえわからず、依存関係を把握することができない。よってバイナリレベル変数解析の手順を提案する。

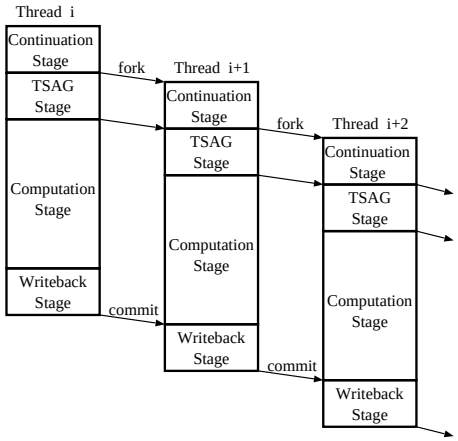

図1 スレッドパイプライニングモデル

この変数解析を行う目的は、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同じ変数に対してのアクセスを検出することである。変数解析の手順と解析情報の活用について説明する。

3.1 変数解析手順

バイナリレベルで変数を解析するには、レジスタの内容およびアクセスされるメモリを解析する必要がある。バイナリレベル変数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
(2) データフロー木の構築
(3) ループイテレーション間依存レジスタの検出
(4) データフロー木の正規化
(5) 同一アドレスのメモリへのアクセスの検出
(6) 仮想レジスタ割り当て

これによって、レジスタ・メモリの内容や操作に関する情報を得ることが可能となるため、メモリ上における変数の位置、ある変数を操作している位置などの情報が得られる。

以降、それぞれの処理について説明する。

3.1.1 レジスタへのインスタンス番号付加

バイナリレベルの場合、同じレジスタであってもコード中の位置によってレジスタの値が同じであるとは限らない。そこでレジスタが更新されたかどうかを明らかにするために、コードを先頭から順にたどり、レジスタへの書き込み命令がある毎にそのレジスタに対してインスタンス番号を付加する。

解析開始点での値に対して#0とし、レジスタが更新される毎に番号を1ずつインクリメントする。同名のレジスタであっても、インスタンス番号が異なっているものに関しては異なった値が入っているものとして判断する。

図2に、アドレス0x4001f0番地を解析開始点としてインスタンス番号を付加した後のコード例を示す。この例でレジスタ$2に着目すると、まずアドレス0x400208

```
4001f0  addiu   $29#1, $29#0, -8
4001f8  sw      $0,    0($29#1)
400200  addu    $5#1,  $0,    $0
400208  lw      $2#1,  0($29#1)
400210  addu    $3#1,  $5#1,  $4#0
400218  addiu   $5#2,  $5#1,  1
400220  addu    $2#2,  $2#1,  $3#1
400228  sw      $2#2,  0($29#1)
400230  slti    $2#3,  $5#2,  100
400238  bne     $2#3,  $0, 400208
```

図2 インスタンス番号付加後のコード例

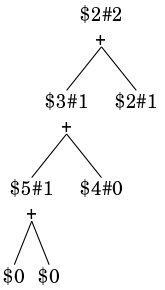

図3 データフロー木の例

番地のロード命令で値が定義されるため、0x400208番地のディスティネーション側にあるレジスタ$2にはインスタンス番号#1が付加される。アドレス0x400220番地においてはレジスタ$2はソース側とディスティネーション側の両方に存在しているが、ソース側は以前の値の使用であるため#1を、ディスティネーション側は新しく定義されるためインクリメントされた#2を付加する。

後述する仮想レジスタもふくめて、全てのレジスタについて同様に番号を付加する。

3.1.2 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データフロー木を構築する。演算をした際に、オペランドのソース側に置かれる定数やレジスタを子、ディスティネーション側に置かれるレジスタを親とするのを基本とし、解析開始点から順にボトムアップで木を構築する。ここで、次項で述べるループイテレーション間依存レジスタ検出処理のために、各節点では要素が最後に利用されたアドレスも保存しておく。

また、ロード・ストア命令について、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。このデータフロー木は同一アドレスへのメモリアクセスの検出の際に用いる。

図2のレジスタ$2#2について生成した木の例を図3に示す。

3.1.3 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、スレッド間依存データは、マルチスレッド実行時にスレッド間で正しく値を受け渡さなければ、正しい結果を得ることができない。ループの誘導変数として用いられるものについては、増減に関する情報がマルチスレッド化する際に必要となる。これらのことから、ループイテレーション間依存レジスタを検出する必要がある。

ループイテレーション内で最初にレジスタの値を定義する際に同じレジスタの以前の値を用いている場合には、そのレジスタはループイテレーション間でフロー依存の関係がある。

イテレーション内で新しい値の定義が行われるレジスタに対して、前項で構築したデータフロー木を葉に向かう方向にたどる。あるレジスタに対応する木について、同名でインスタンス番号が小さいレジスタをイテレーション内で利用しており、かつそのインスタンス番号が小さいレジスタの元になる値がイテレーション外で定義されているという条件に当てはまる場合に、このレジスタをループイテレーション間依存レジスタとして検出する。

次項の正規化以降の処理において、ここで検出された依存レジスタ以下の部分木に対しては処理を行わず、依存レジスタは葉として扱う。また正規化を行ってしまうと、データフロー木が変形して依存の把握ができなくなるため、この段階で依存レジスタの検出を行う必要がある。

3.1.4 データフロー木の正規化

定数が分散して配置されていたり、複数の要素についての同じ演算の順序が入れ替わっていたりなど、データフロー木には、同じ内容を表していても違う形となる場合が多く存在する。これらの場合にもデータフロー木の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。

データフロー木が以下の条件をすべて満足する状態にあるとき、正規化された状態と定義する。

(1) 簡約可能な演算は存在しない
(2) 数式で表現した場合に完全に展開された形である
(3) 同じ親を持つ子は一定の順序関係に従って整列している
(4) 数式で表現した場合の同類項にあたる部分が存在しない

データフロー木を正規化された状態にするために、以下の処理を木の変形が起こらなくなるまで反復実行する。

(1) 不要な部分の削除
加算されている定数0や乗算されている定数1、0を乗算されている部分全体など、不要な部分を削除する。
(2) 計算可能な値の評価
定数どうしの演算など、計算可能な値は評価してまとめる。
(3) 複数段で表されている同一演算子の簡約化
図3における加算のように、複数段で構成されている同一の演算を、できるだけ低い木となるよう簡約化する。図3の加算を簡約化したものを図4に示す。
(4) 乗算・加算の展開
乗算の子に加算がある木は、数式で表現したときに整式の積となる部分が存在することになる。そのため、多項式の展開と同様にして乗算・加算の展開を行う。例えば図5の木を数式で表すと

$(3+z)(x+y)$

となるが、これを展開された形である

$3x+3y+xy+xz$

に相当する図6の木に再構築する。

(5) 要素の整列

交換法則が成り立つ演算である親を持つ子は、一定の順序関係にしたがって整列する。

(6) 同類項の要約

数式で表現したときに、定数のみを係数としたときの同類項が存在する場合、同類項をまとめることで木を簡約化する。例えば木を数式で表したときに

$3xy+5xy$

であるものと

$2xy+6xy$

であるものは、ともに

$8xy$

で表される木とする。

以上の処理を木に対して適用可能な限り繰り返せば、データフロー木は正規化された状態となる。

3.1.5 同一アドレスのメモリへのアクセスの検出

変数がメモリ上に置かれている場合、同一の変数を操作している箇所は、バイナリコード上では同一のメモリを操作している箇所となる。この情報を得るため、同一アドレスのメモリへアクセスをしている箇所を検出する。

バイナリコード上で同一のメモリを操作している箇所を特定するには、間接指定に用いられるレジスタとオフセット値の情報を用いて、対象が同一であるアドレス指定を検出する。

ロード・ストア命令で、アドレスを指定するために用いているレジスタとオフセット値から構成されたデータフロー木を総当たりで比較し、同一の内容を持つデータフロー木であった場合に、アクセス対象とされているメモリは同一であると判断する。データフロー木は正規化されているため、同一の内容であれば同一の形である。よってデータフロー木の形を比較して同一のものを検出する。

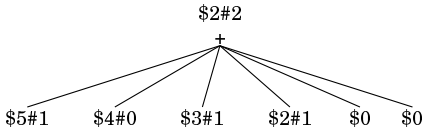

図4 同一演算子の簡約化の例

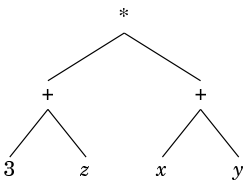

図5 展開前の木

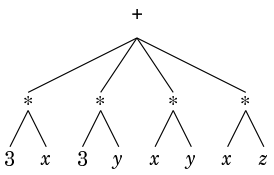

図6 展開後の木

3.1.6 仮想レジスタ割り当て

以上までの処理では、メモリを介するデータは解析することができない。そこで仮想レジスタを導入することでメモリ上のデータを解析可能とする。

メモリへの操作を仮想レジスタへの操作とみなして再びレジスタへのインスタンス番号付加からの解析処理を行うことで、メモリに格納される値を解析することが可能となり、より深い解析ができる。

対象アドレスが固定であるメモリアクセスが存在する場合には、対象となるメモリを仮想的なレジスタとみなしてロード・ストア命令をレジスタ間転送命令に置き換え、仮想レジスタを実現する。

例として図2のコードの場合について説明する。レジスタ$29はスタックポインタであるとすると、同一手続き内においてスタックポインタは基本的に固定であるため、アドレス0x4001f8番地、0x400208番地、0x400228番地での操作対象となるアドレス(0+$29#1)番地は固定であると考えることができる。そこで、アドレス(0+$29#1)番地のメモリを仮想レジスタ$V1とみなし、ロード・ストア命令をレジスタ間転送命令に置き換えたものが図7である。このコードの場合では、仮想レジスタ割り当て後のコードに対して再び解析処理を行うことで、仮想レジスタ$V1に割り当てられたアドレス(0+$29#1)番地のメモリがイテレーション間依存変数であるということがわかる。

3.2 変数解析情報の活用

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(仮想レジスタを含む)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、誘導変数として扱う。

従来、メモリ上に誘導変数が置かれている場合は、どのアドレスのロード・ストア命令が誘導変数に対してであるのかが判断できず、位置や増減などについてバイナリレベルでは検出が困難であった。誘導変数が検出できない場合、前のイテレーションでの値が決定するまで次のイテレーションは実行を開始することができない。そのため並列度を大きくすることができず、ほとんどマルチスレッド化前のコードと変わらない実行時間となってしまうか、マルチスレッド化が不可能

```
4001f0  addiu  $29#1, $29#0, -8
4001f8  mov    $V1#1, $0
400200  addu   $5#1,  $0,    $0
400208  mov    $2#1,  $V1#1
400210  addu   $3#1,  $5#1,  $4#0
400218  addiu  $5#2,  $5#1,  1
400220  addu   $2#2,  $2#1,  $3#1
400228  mov    $V1#2, $2#2
400230  slti   $2#3,  $5#2,  100
400238  bne    $2#3,  $0,    400208
```

図7 仮想レジスタ割り当て後のコード例

表1 シミュレーションパラメータ

スレッドユニット	4命令同時実行 out-of-order 実行
1次キャッシュ (命令)	direct-map 16KB (ラインサイズ32B) レイテンシ1クロック
1次キャッシュ (データ)	4-way set-associative 16KB (ラインサイズ32B) レイテンシ1クロック
2次キャッシュ (命令・データ混在)	4-way set-associative 256KB (ラインサイズ64B) レイテンシ6クロック
メモリ	レイテンシ18クロック
メモリバッファ	512B
スレッドユニット間 通信ポート	レイテンシ1クロック

となる。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誘導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でも、効率の良いマルチスレッドコードを生成することが可能となる。

また誘導変数の変化を把握することで、この情報をもとにイテレーション毎に変化する対象アドレスの値を計算し、数値計算系アプリケーションでの配列の操作などの、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

前節で説明したバイナリレベル変数解析の有効性を検証する。従来バイナリレベルではマルチスレッド化困難であったアプリケーションについて変数解析・マルチスレッド化を行い、速度向上比 (=逐次実行時のサイクル数/マルチスレッド実行時のサイクル数) を計測して実行性能の評価を行う。

4.1 評価環境

バイナリレベルマルチスレッド化実現のためのバイナリコードの変数解析を行うために、第3節のバイナリレベル変数解析の手順を実装した。変数解析には、このバイナリレベル変数解析プログラムを用いる。

アプリケーションプログラムの実行には、SimpleScalarをベースにした、スレッドパイプライニングモデル[3]を実現するアーキテクチャシミュレータ SIMCA[4]を用いる。SIMCAのシミュレーションパラメータは表1に示すものを用いる。

4.2 評価対象

評価対象アプリケーションは、SPECfp95の101.tomcatvとし、対象ループは、実際に演算処理を行っている6つの最内ループ (#1〜#6) とした。SPECfp95のソースコードはFORTRAN77で記述されているため、まずf2c FORTRAN77 to C トランスレータ (version 19940927) によって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, 最適化オプション-O2) を用いて対象とするバイナリコードを生成する。

表2 ループ#1の誘導変数に対する操作

命令のアドレス	コード
400f08	sw $22, 0($19)
401168	lw $4, 0($21)
401460	lw $2, 0($21)
4014d0	lw $3, 0($21)
401848	lw $2, 0($21)
401888	lw $2, 0($21)
401898	sw $2, 0($21)

101.tomcatvは、2次元のメッシュを境界に沿った形で生成するプログラムである。この101.tomcatvのバイナリコードでは、元のソースコード上でのメインルーチンのサイズが大きいことと、f2cでの変換による影響によって、ほとんどの変数がメモリ上に置かれている。メモリ上に置かれている全ての変数は、レジスタ間接でアドレスを指定するロード・ストア命令によって操作されるため、変数解析をせずに、どの変数に対する操作なのかを判断することはできない。

101.tomcatvの場合では、いずれのループにおいても、イテレーションを開始するために必要な誘導変数がメモリ上に置かれている。そのため、バイナリレベルでのマルチスレッドコード生成を、変数解析なしで行うことは不可能であった。そこでバイナリレベル変数解析により誘導変数を操作している箇所を特定し、バイナリレベルでのマルチスレッド化を行った。

4.3 評価結果

gcc(-O2で最適化)によって生成された101.tomcatvのバイナリコードをバイナリレベル変数解析プログラムに入力して変数解析を行った結果、対象としたいずれのループについても、イテレーション実行開始に必要な誘導変数を除いたイテレーション間での依存関係は検出されなかった。

誘導変数については、いずれのループでも同じアドレス ($sp#1+0xe11f10) のメモリを用いていることがわかった。また、1イテレーションあたりでの増減値はすべて+1であった。参考として、ループ#1(アドレス範囲0x400f58〜0x4018af) の誘導変数にあたるメモリにロード・ストアを行っていると変数解析によって検出された命令のアドレスを表2に示す。

この表での0x400f08番地におけるアドレス ($19+0) と、他におけるアドレス ($21+0) は、実際には同じアドレスを指している。このように、異なったレジスタをアドレス指定に利用して同じメモリを操作していることがわかる。

誘導変数の場所と増減の情報を用いて実際にマルチスレッド化を行い、SIMCAを用いて並列実行可能

表3　101.tomcatv 各ループの命令数

ループ	#1	#2	#3	#4	#5	#6
命令数	299	18	70	38	57	40

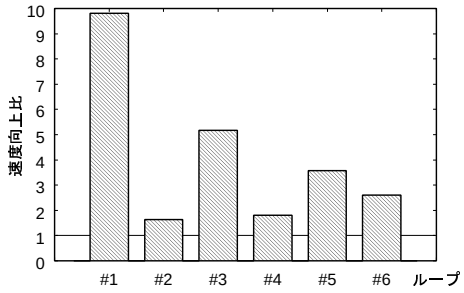

図8　101.tomcatv での各ループの速度向上比

スレッド数16で速度向上比を計測した結果を図8に示す。それぞれ逐次実行時と比較して9.803倍、1.642倍、5.177倍、1.800倍、3.583倍、2.611倍の速度向上が得られている。従来のバイナリレベル変数解析手法を適用していない方式では、101.tomcatv の各ループはバイナリレベルでマルチスレッド化することが不可能であった。しかし、本手法を適用することで得られた変数解析情報を活用することで、ループの誘導変数を検出し、バイナリレベルマルチスレッド化を実現することが可能となり、速度向上を達成したことがわかる。

どのループも逐次実行時より高速化しているが、特にループ#2, #4の速度向上比が他のものよりも小さいことがわかる。表3にマルチスレッド化前バイナリコードにおけるループ1イテレーションあたりの命令数を示すが、この表によると、ループ#2や#4は1イテレーションあたりの処理サイズが小さい。そのためにマルチスレッド化した際の Computation ステージに相当する部分のコードサイズが小さくなる。この場合はスレッド制御にかかるオーバヘッドの影響を受けるため、十分な並列度を得ることができない。速度向上比が大きくならない原因はここにある。

本変数解析手法の適用によって依存関係が明確となったため、ループアンローリング等の手法が適用可能となる。これらの手法により処理サイズを十分大きくすることで、スレッド制御にかかるオーバヘッドの影響を小さくすることが可能である。また、今回はマルチスレッド化を行う際に最適化処理を行っていないが、マルチスレッド化やループアンローリングを行う際に、解析情報を用いて適切な最適化処理[5),6)]を行うことで、より大きな速度向上を得ることが可能である。

5. おわりに

本稿では、バイナリレベルで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変数解析の手順を示した。実際のアプリケーションのバイナリコードへ本手法を適用して解析することで、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、実行速度の向上を図った。

評価の結果、バイナリレベル変数解析の結果を利用してマルチスレッド化し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最大9.8倍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変数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。また、マルチスレッド化時に変数解析情報を活用して最適化処理を行うことで、アプリケーションの実行速度をより向上させることができると考えられる。

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) 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, “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,” 情報処理学会第66回全国大会講演論文集, 講演番号6T-1, pp.1-125〜1-126, 2004.

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

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

5) D. F. Bacon, S. L. Graham and O. J. Sharp, “Compiler Transformations for High-Performance Computing,” ACM Computing Surverys, Vol. 26, No. 4, pp.345〜420, Dec. 1994.

6) 阿久津 徳寿, 佐藤 智一, 月川 淳, 大津 金光, 横田 隆史, 馬場 敬信, “バイナリレベルマルチスレッド化におけるスレッドコード最適化,” 情報処理学会第66回全国大会講演論文集, 講演番号6T-3, pp.1-129〜1-130, 2004.