Vol.2009-ARC-184 No.8
2009/8/4

実行時データ依存解析による ループ階層構造に着目した並列性抽出

佐藤幸紀[†1]　中村維男[†2]

大規模・複雑化するアプリケーションを高度に並列化することを目的として、既存の逐次コードをマルチコアプロセッサ上でトランザクションメモリあるいはスレッドレベル投機実行を用いて逐次的な実行結果を保証しつつ並列に動作させるというランタイム自動並列化が注目を集めている。このランタイム自動並列化を実現するためには発見的で解空間の広い並列化対象から的確に並列化候補を選択する必要がある。本論文では、実行時プロファイリング技術を用いてバイナリコード実行時にループ領域毎のデータ依存関係を調べ、並列性を抽出する手法を提案する。提案する手法は、ループ階層構造を利用し参照した全てのメモリアドレスについて依存関係を把握するために必要な情報を効率的に記録する。評価環境を構築し評価を行った結果、ループ領域に着目したデータ依存解析はデータ依存関係を効率的に把握する上で有効であること、さらに、潜在的な DOALL ループを検出できることを確認した。加えて、評価対象を構築する上で利用するコンパイラや命令セットの構成を変化させて評価を行った結果、抽出される潜在的な DOALL ループは利用するコンパイラや命令セットと密接に関係があることがわかった。

Extracting parallelism in nested loop structures using run-time data dependency analysis

Yukinori Sato[†1] and Tadao Nakamura[†2]

Run-time parallelization based on transactional memory or thread-level speculation is one of techniques to realize highly parallelized processing while guaranteeing the same results as sequential execution. To realize effective run-time parallelization, we have to seek appropriate code regions for parallelization in a program. In this paper, we propose a method of data dependence analysis, which focuses on using loop regions in binary code and captures dynamic behavior of control flows and memory references effectively using dynamic binary instrumentation technique. We implement our data dependence analysis scheme and evaluated it. The results show that our scheme using loop region information can dramatically reduce the number of detected data dependencies compared with the conventional address based scheme. The results also show that potential DOALL loops can be extracted using our scheme and whether they can extract or not is strongly related to the ISA of the CPU and compiler we use.

1. はじめに

マイクロプロセッサの設計の主流がマルチコアプロセッサに移行していることに伴い、プログラムに内在する並列性を検出し並列なハードウェア上で効果的に処理する並列処理技術に高い関心が集まっている。近い未来に登場するであろうチップ内に数 10 から数 100 の CPU コアを実装するメニーコアプロセッサや、近年注目されている SIMD、FPGA といったアクセラレータを効果的に利用するためにはそのハードウェアの理論性能に近い性能を持続的に引き出すことを可能とする現在の水準よりも高度な並列処理技術が求められると予想されている。

メニーコアプロセッサやアクセラレータといった強力な演算能力を備える並列処理エンジンから持続的に理論性能に近い性能を引き出すためには、アプリケーションの実装毎に変化する並列化するべき対象を的確に見つけ出すこと、そして、その部分を並列実行する適切な手段を並列処理エンジンのマイクロアーキテクチャを意識しつつ決定することが必要である。このような高度な並列化は一般的に発見的で解空間の広い困難な作業であることが多い。さらに、このような高度な並列化に由来する生産性低下の問題に加えて、年々進行する実アプリケーションプログラムの大規模・複雑化に由来するプログラミング生産性低下の問題も同時に考える必要があり、高度な並列化を生産的に達成する手法の確立が強く求められている。

既存の逐次コードをマルチコアプロセッサ上で並列に動作させるというランタイム並列化に関して様々な技術が提案されている[1)–3)]。ランタイム並列化は大規模・複雑化するアプリケーションに関してもユーザーの明示的な並列性記述なしに高度に並列化することが可能であり、生産性な並列化手法であるといえる。ランタイム並列化を行う上で重要な動作として、並列化の対象として適切な部分を推定することと、並列化対象部分を並列に処理した際

†1 北陸先端科学技術大学院大学 情報科学センター
Center for Information Science, Japan Advanced Institute of Science and Technology
†2 慶應義塾大学
Keio University

の結果が逐次処理の際の結果と一致することを保障することがある。後者に関しては、トランザクションメモリの機構、あるいはスレッドレベル投機実行の機構など広く提案されている機構を用いてプログラムが本来持つ逐次的性質を超える並列処理を逐次的な実行結果を保証しつつ実行することは可能である。しかしながら、前者に関しては並列実行するべき適切な部分を検出したことを前提としている議論が大部分を占めている。すなわち、プログラマが明示的に与える並列部分の情報に頼っているという側面が多く、高度な並列化のために必要となる発見的で解空間の広い並列化対象から的確に並列化候補を選択する技術としては確立されていない。

プログラムから適切な並列化対象を検出するために実行時プロファイリング技術を用いて実行時にデータ依存関係の解析を行うというアプローチがある。実行時プロファイリングを用いたデータ依存解析によりコンパイル時には把握が困難であったポインタによる間接メモリアクセスも詳細に解析することが可能となる。このような詳細なデータ依存解析により、適切かつ効果的に並列化の対象を抽出することが可能となるといえる一方で、参照した全てのメモリアドレスについて依存関係を把握するために必要な情報を的確に記録する効率的手法が求められている。

そこで、最終的に必要となる並列化の対象となる部分の代表的な粒度となるループ構造に着目して、検出したループ構造単位でデータ依存の有無を管理することを提案する。提案手法により、実行時プロファイリングにより検出したループ階層情報とデータ依存関係を互いに結びつけることにより命令レベル並列性よりも粗粒度なループレベルの並列性をバイナリコード実行時に抽出することを実現する。

本論文では、JIT コンパイラ技術を用いた実行時プロファイリングツールである Pin[4] を利用してコンパイル済みの実行コードを実行させる際に実行時にループ階層構造とデータ依存関係をプロファイリングする機構を構築し、実アプリケーションにおけるループ構造とデータ依存関係の関係性を調べる。本手法はコンパイル済みの実行コードからバイナリコード実行時に並列化対象を推定する為プログラマが並列性を明示する必要がない高い並列効率と生産性を両立する並列処理の方式であると考えられる。

本論文の構成は以下の通りである。2 節ではデータ依存解析に関して既存の手法とその問題点を述べる。3 節では提案するループ階層構造に着目したデータ依存解析手法を説明する。4 節では提案する手法について基礎的評価を行った様子について述べる。5 節は結論である。

2. 既存のデータ依存解析手法

データ依存解析は並列ループの抽出の鍵となる技術であり、伝統的に FORTRAN に代表されるような言語における添え字付きの配列式について広く行われてきた。しかしながら、C や C++のようなポインタを広く利用するような言語に関しては配列の添え字に限定した依存解析では不十分であり、ポインタの動的な振る舞いを含めたデータ依存関係を解析する必要がある。

エイリアスプロファイリングはポインタが引き起こすデータ依存関係を検出するために使われている技術である[5]。エイリアスとはプログラム中のメモリオブジェクトの別名であり、エイリアスの示すアドレス空間の一連の組に対応する。従って、エイリアスについて解析を行うエイリアスプロファイリングにおいてはポインタの指し示すメモリアドレスをそれぞれ独立して取り扱うのではなく参照するメモリアドレス空間の一連の組を 1 つのブロックとして扱う。そのため、エイリアスプロファイリングにより得られる情報は全てのメモリアクセスをアドレス単位でプロファイリングすることにより得られる情報よりも一般的に精度が低くなる[6]。

メモリアクセスをアドレス単位でプロファイリングするデータ依存解析はエイリアスプロファイリングよりも細かな粒度であるアドレスを単位としてメモリのあいまいさがないように解析を行う。しかしながら、このようなデータ依存解析により抽出されるデータ依存関係の数はエイリアスプロファイリングよりも数桁大きくなることが一般的である。従って、データ依存関係の動的な振る舞いを精度や抽出数の観点から効率の良い方法を実現することが求められている。

3. 提案するデータ依存解析

3.1 提案手法の概要

データ依存を正確に解析するためには全てのメモリ参照を取り扱う必要がある。一方で、プログラムの実行中において膨大なメモリ参照が行われるため、単に全てのメモリ参照の履歴を保持するのではなくメモリ参照に対してのデータ依存関係を効率的な手法で記録する必要がある。さらに、データ依存関係の解析結果は本研究で目的としているような並列実行可能な部分を推測しトランザクションメモリやスレッドレベル投機実行により投機的な並列処理を行うことを想定する場合、投機的処理の利得を精度よく計算するために必要十分な情報量を持ち、並列化のために理解しやすい形式である必要がある。

Vol.2009-ARC-184 No.8
2009/8/4

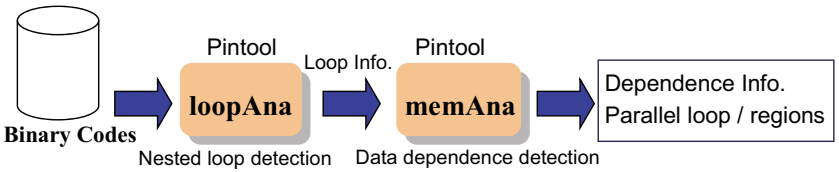


図 1 ループ階層構造に着目したデータ依存解析のフロー

このような目標を達成するために、我々はループ領域間のデータ依存関係に着目する。ループ領域の情報を用いてメモリ参照によるデータ依存関係をそのメモリアクセス命令が参照するアドレスに対して最後に書き込みを行ったメモリアクセス命令があるループ領域に依存があるとして取り扱う。さらに、データ依存関係をループ領域ごとに管理することにより、最終的にはループ領域間に並列性があるかどうかを理解することが可能となる。

例えば、あるメモリアドレスにあるインスタンスがループ領域 A により読み込まれ、そのメモリインスタンスがループ領域 B により生成されたものである場合、本手法ではループ領域 B をループ領域 A の依存のある領域として取り扱う。これは動的なコード実行に対するループに着目したエイリアスというようにも考えることが可能であり、メモリアクセスに関してはアドレスの単位での正確な把握が可能となる。

本論文では、提案するデータ依存解析を実現するためにループ構造検出（loopAna）とメモリアクセス検出（memAna）の 2 つのステージから構成されるプロファイリング機構を Pin[4] を用いて構築した。図 1 に本論文で行ったループ階層構造に着目したデータ依存解析のフローを示す。

Pin は just-in-time (JIT) コンパイラ技術を用いて動的にバイナリコードに補償コードを挿入し実行時プロファイリングを行うツールであり、仮想マシン（VM）と VM に命令を供給する機構から構成される。VM のための命令の生成は実行時に JIT コンパイラにより行われる。JIT コンパイラはプロファイルのための補償コードをバイナリコードの所望の場所に挿入する。JIT コンパイラにより生成された補償コード入りのコードはコードキャッシュと呼ばれる特別な領域に保持され、VM はコードキャッシュから命令をフェッチすることにより補償コード入りの命令が VM 上で実行される。Pin の API により記述された Pintool と呼ばれる補償コードを本論文においてはループ構造検出（loopAna）とメモリアクセス検出（memAna）のためにそれぞれ用意した。

ループ階層構造に着目したデータ依存解析の最初のステージであるループ構造検出

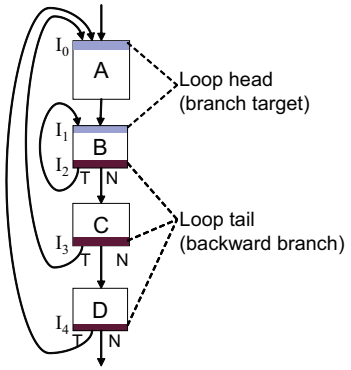


図 2 階層構造ループの例

（loopAna）は、我々が開発したループ階層構造を検出する機構を用いた[7]。プロファイリングによるループ構造検出の後、検出されたループ階層構造を用いてバイナリコード中のループ領域の位置を示すマーカーを生成する。生成したマーカーは次のステージであるメモリアクセス検出（memAna）の際にループ領域情報を指し示す為に利用される。プロファイリングによるメモリアクセス検出のステージではメモリ参照を行う命令に対応するループ領域によりメモリ参照の依存の履歴を記録する。メモリアクセス検出の後、ループ領域間の依存関係の有無を確認し、最終的に並列に実行できる可能性が高いループを得る。

3.2 ループ解析ステージ

本論文で用いた動的にループを検出する手法の概要を説明する。本手法は入れ子となっているループ構造や関数呼び出しを検出することが可能である[7]。

図 2 にループ階層構造の例を示す。図中の箱は基本ブロックを示し矢印はコントロールフローを示す。ここで基本ブロックとは実行時に生成される動的な基本ブロックを示している。ループ構造において常にループの最後尾にある後方分岐命令は一連の命令実行のフローをサイクリックに行う役割を果たすループにおいて最も意味を持つ命令である。加えて、後方分岐命令の飛び先のターゲットとなる命令もループの先頭となる命令であり、ループにおいて後方分岐命令と同様に重要な意味を持つ命令となる。

本手法においてはループを構成する最後尾の命令（ターゲットアドレスを同一の関数内に持つ後方分岐命令）、先頭命令（最後尾にある後方分岐命令のターゲットアドレス）、さらに、関数呼び出しやそのリターンを行う命令の位置をそれぞれ検出し、入れ子となっている

関数とループの構造を解析する。ループの先頭命令と最後尾の命令の位置をそれぞれ固有にもつ組を独立したループとみなし、ループ構造を検出する。図 2 の例では、先頭と最後尾の命令アドレス組で $(I_0, I_4), (I_0, I_3), (I_1, I_2)$ という 3 つのループが検出される。さらに、関数呼び出しやリターン命令の位置も含めてループの特徴となる命令の位置を命令アドレスの順にソートすることによって、ループ階層構造や関数とループが入れ子される構成を把握することが可能となる。

ループプロファイリングの過程において、直前のループ最後尾命令が実行されてから該当するループ最後尾命令が実行される間に実行された命令数を計測し、この命令数をループの 1 つの反復中で実行される命令数として取り扱う。さらに、実行時間が動的に実行された命令数に比例すると仮定すると、各ループにおいて実行された命令の合計を求めることによりループ毎の実行時間を推定することが出来る。そこで、本論文では各ループにおいて実行された命令数を用いて、そのループがホットループかどうかの判断を行う。

3.3 データ依存解析ステージ

我々の提案するループ階層構造に着目したデータ依存解析は以下の過程により行われる。まず、検出されたループ階層構造を用いてバイナリコード中のループ領域の位置を示すマーカーを生成する。次に、メモリアクセスをプロファイリングするのための補償コードに現在実行しているループを示すマーカーの情報を組み合わせ、依存関係をループ階層構造に基づき表現するメモリアクセス検出機構を用意する。用意したメモリアクセス検出機構を用いてプロファイリングを行い、最終的に並列に実行できる可能性が高いループ領域を得る。

データ依存解析における各過程の詳細を以下に説明する。ループ領域を認識するためのマーカーはループ解析ステージにより検出されたループ構造を読み込み作成する。図 3 にマーカーとマーカーの挿入される実際の命令アドレスとマーカーにより識別されるループ領域を示す。マーカーをループ領域の先頭とループ領域の最後に挿入することによりループ領域を識別する。マーカーにより識別されるループがホットループである場合は、ホットループであるという情報を持つマーカーを挿入する。マーカーによりバイナリコードの領域はループ領域、ホットループ領域、ループの外側に分類される。入れ子のループ構造である場合、その領域の最内ループをループの代表領域とする。マーカーを用いてバイナリコード上における領域を分割することにより、バイナリコード上の全ての実行される命令は 1 つの領域に対応する。

次に、マーカーが挿入されたバイナリコードを用いてデータ依存解析のための実行時プロファイリングを行い、メモリアクセスに関する情報を順次記録していく。メモリアクセスに

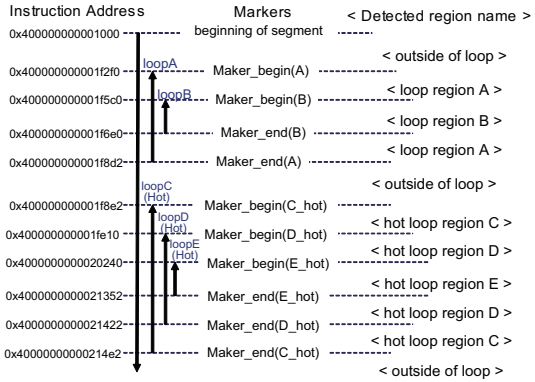


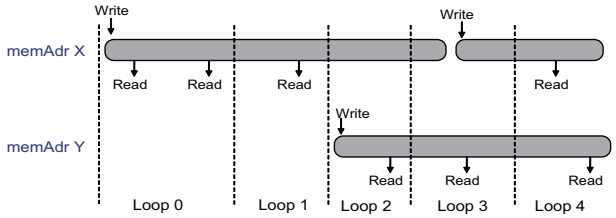
図 3 ループ領域の表現法

関する情報にはアクセスするメモリアドレス、アクセスの種類（リード、ライト）、メモリアクセス命令が位置するループ領域などが含まれる。本論文では膨大な量のメモリアクセスに関する情報を効率的に取り扱うためにホットループの領域が依存しているループ領域にのみ着目しデータ依存を調べる。

ループ領域の並列性を推定する上で、メモリコンシステンシモデルによりどのようなメモリ参照の順番がデータ依存がある関係になるかが変化するためメモリコンシステンシモデルは重要な役割を持つ。例えば、Sequential Consistency モデルにおいてはプロセス内の全てのメモリ操作の順序はそのプロセスが決めた逐次実行の順序に従うことが要求される[8]。Release Consistency モデルにおいては先行するリードと後続のライトや、先行するリードと後続のリードの順番に関する制約を和らげることを許容している場合もある[9]。

本論文では、(1) allAccessRegion ポリシという同一のアドレスへのメモリ参照をすべて依存があるとみなす Sequencial consistency モデルを意識したデータ依存解析、(2) lastWriteRegion ポリシという各メモリアドレスへ最後に行った書き込み操作のみがその後の読み込み操作に依存するとみなすデータ依存解析の 2 つを実装する。

図 4 にアドレス X と Y へのメモリアクセスとアクセスを行っているループ領域の例を示す。この例においてループ領域 1 はアドレス X にアクセスしている。アドレス X に関してはループ領域 0, 1, 3, 4 がアクセスを行っているため、allAccessRegion ポリシを用いるとループ領域 1 はアクセスがある全てのループ領域と依存する。lastWriteRegion ポリシを用いると、ループ領域 1 のリードはループ領域 0 におけるライトにのみ依存していることと



Regions	Loop 1	Loop 4
(1) allOfAccess	Loop 0, 1, 3, 4	Loop 0, 1, 2, 3, 4
(2) lastWrite	Loop 0	Loop 2, 3

図 4 ループ領域 1 とループ領域 4 におけるデータ依存

なる。ループ領域 4 に関してはアドレス X とアドレス Y の両方にアクセスを行っている。アドレス Y に関してはループ領域 2, 3, 4 がアクセスを行っている。allAccessRegion ポリシを用いるとループ領域 4 はアドレス X とアドレス Y にアクセスのある全てのループ領域と依存していると解析される。一方、lastWriteRegion ポリシを用いると、アドレス X に関する最後のライトのあるループ領域 3 とアドレス Y の最後のライトであるループ領域 2 と依存していると解析される。

allAccessRegion ポリシの場合、RAW 依存に加えて WAR 依存、WAW 依存も検出することが可能であるが、データ依存関係を過大評価しているという側面もある。lastWriteRegion ポリシの場合、RAW 依存のみに着目し、その他の依存は新しいメモリインスタンスを生成するなど何らかの形で回避すると仮定している。ループ領域における潜在的な並列性を推定するという観点からは lastWriteRegion ポリシが望ましいと考えられる。

4. 評　　価

4.1 評 価 手 法

提案するループ階層構造に着目したデータ依存解析の有効性を評価するためにループ構造検出（loopAna）とメモリアクセス検出（memAna）の 2 つのステージから構成されるプロファイリング機構を Pin[4] を用いて構築し、並列性があると推定されるループの抽出を行う。並列性抽出の上での命令セットやコンパイラの影響を調べるために Pin を IA32 と IA64 命令セットのそれぞれに構築した。

IA32 の環境においては Pin version 2.5 IA32 Linux 用を利用しプロファイリング機構を構築した。OS は CentOS 5.2 Linux、CPU は Intel Core2 Quad を使用した。コンパイラは gcc 4.1.2 を使用し、-O のオプションによりソースコードのコンパイルを行った。

IA64 の環境においては Pin version 2.6 IA64 Linux 用を利用し、Intel Itanium2 を CPU として持つ SGI Altix 4700 上で動作している SuSE Enterprise Sever 10 Linux を使用した。コンパイラは gcc 4.1.2 とインテルコンパイラ icc 10.0 の 2 種類をを使用し、それぞれ-O のオプションによりソースコードのコンパイルを行った。なお、IA64 の命令セットにはループカウンタを利用した定数ループのための分岐命令である br.cloop 命令やソフトフェアパイプライニングのための br.ctop 命令などループに関する様々な命令が定義されている。

評価実験を行うためのベンチマークプログラムとして、NAS Parallel Benchmark v3.3 の Serial 版を使用した。NAS Parallel Benchmark においては IS、DC というプログラムは C で記述され、残る BT、CG、EP、FT、LU、MG、SP、UA は FORTRAN により記述されている。解析には S データサイズを利用した。プロファイリングの対象には、ソースコードをコンパイルすることによる実行コードに加えて、ソースコードよりリンクされる実行中に読み込まれるライブラリも含まれる。

本論文では、ループの並列化による速度向上を目指すという観点から実行時間全体に占める割合の高いホットループを基準として、ホットループが依存するループ領域を調べた。ホットループの抽出は loopAna ステージにおいて行うとした。loopAna ステージにおいて各ループの実行する命令数が全体に占める割合を求め、実行割合が 0.1%を超えるループをホットループとするとした。memAna ステージにおいてはホットループと依存のあるループ領域を記録していくことによりけるデータ依存解析を行っていくが、ループと判断されなかった部分と依存がある場合についてはループ外の領域と依存があるという一元的な取り扱いを行った。

memAna ステージにおいてはデータ依存解析を行うデータ幅として 4 バイトを単位とするとした。また、本論文においてはメモリを解したデータ依存のみを取り扱ったため、レジスタを解したデータ依存は検出されないことに注意しなければならない。レジスタを解したデータ依存は実行時プロファイリングを利用した解析よりもコンパイラによる静的な解析においても効率的に取り扱うことが可能と思われるため、コンパイラによる静的な解析との連携も視野に入れる必要がある。

4.2 データ依存解析結果

表 1 にループ解析ステージにて検出されたループを示す。検出されるループの数はアプリ

表 1 検出されたループの数

prog.	ia32.gcc	ia64.gcc	ia64.icc
bt	730	717	598
cg	604	600	530
dc	537	561	631
ep	561	561	526
ft	578	590	526
is	364	327	358
lu	723	717	601
mg	597	590	580
sp	793	790	654
ua	1010	1024	1280

表 2 検出されたホットループの数

prog.	ia32.gcc	ia64.gcc	ia64.icc
bt	46	40	33
cg	16	16	14
dc	31	36	44
ep	2	4	2
ft	16	18	16
is	8	9	11
lu	54	55	42
mg	30	28	37
sp	100	98	89
ua	93	100	93

ケーションプログラムの特性に大きく依存しているため実行環境の変化によるループ数の相違はそれほど見られなかった。しかしながら、特に、ia32 と ia64 環境において gcc を用いた場合、検出されるループ数の傾向はほぼ同等であるということがわかった。しかしながら、ia64 環境における gcc と icc を比較するとコンパイラによる相違は若干見られた。

表 2 にそれぞれの実行環境において検出されたホットループの個数を示す。結果より実行環境の変化による相違はほとんど見られないことを確認した。このことは、ホットループとして検出されたループ数はアプリケーションプログラムの特性に大きく依存していることが理由として考えられる。

表 3 にさまざまなポリシに基づき検出されたデータ依存の数を示す。本実験では、メモリアクセスを行ったアドレスの数（accessAdr）、allAccessRegion ポリシにおけるデータ依存の数（allAccess）、lastWriteRegion ポリシにおけるデータ依存の数（lastWrt）をそれぞれの環境にて測定した。

以前より行われてきたメモリアクセスをアドレス単位でプロファイリングするデータ依存解析においては accessAdr の列に示された全てのアドレスについてのデータ依存を解析することが必要となる。さらに、アドレス単位のプロファイル結果を並列部分の推定に利用するために非常に膨大な量の依存関係を解析する必要があることが理解できる。

結果より allAccessRegion ポリシや lastWriteRegion ポリシにより検出されたデータ依存の数はメモリアクセスを行ったアドレスの数よりも大幅に小さいことがわかる。このことより、ループ領域に着目したデータ依存解析は膨大なメモリアクセスの情報を効率的に管理しているということが理解できる。さらに、基準とするループ領域がどのループ領域と依存関係があるかを調べることにより、並列関係のあるループ領域の推定が容易に行える。

表 3 各環境により検出されたデータ依存の数

(a) ia32.gcc

prog.	accessAdr	allAccess	lastWrt
bt	62158	2265	278
cg	271514	227	66
dc	3968703	199	46
ep	150102	28	11
ft	1338267	268	74
is	149212	28	18
lu	63648	1820	348
mg	150961	1107	247
sp	64551	6147	597
ua	643649	9191	1411

(b) ia64.gcc

prog.	accessAdr	allAccess	lastWrt
bt	65242	1196	188
cg	239298	117	49
dc	3987932	111	26
ep	158664	5	4
ft	1345897	118	54
is	152056	30	21
lu	71313	1202	238
mg	158569	348	167
sp	67308	4053	452
ua	651446	6034	1280

(c) ia64.icc

prog.	accessAdr	allAccess	lastWrt
bt	72760	1704	597
cg	256188	528	135
dc	4195287	646	211
ep	161205	107	10
ft	1369851	529	130
is	159862	165	8
lu	81023	1429	590
mg	161842	1225	495
sp	78267	5054	1508
ua	666872	6808	91

すなわち、検出したデータ依存関係のあるループ領域の結果よりループ階層構造となるループとの依存関係の有無も判定することが可能である。

allAccessRegion ポリシと lastWriteRegion ポリシを比較した場合、allAccessRegion ポリシにより検出される依存関係の数は lastWriteRegion ポリシと比べて一桁以上多いということが確認できる。このことより、allAccessRegion ポリシが依存関係を多大評価しているとも考えることができ、投機的実行による並列化の可能性を検討する上では lastWriteRegion ポリシのほうが適していると考えることが可能である。

それぞれの実行環境による検出されたメモリアクセスはおおよそ同等の数が検出されていると考えられるが、ループ領域に着目し検出されたデータ依存関係の数にはばらつきがあることが確認できる。表 3 において (a) と (b) を比較することにより同一のコンパイラを利用してもターゲットとなる命令セットが異なる場合、検出される依存関係の数が異なることが確認できる。(b) と (c) を比べた場合、同一の命令セットを持つ CPU のためのコードで

表 4 検出された潜在的に DOALL の可能性があるループ数

prog.	ia32.gcc	ia64.gcc	ia64.icc
bt	26 (57%)	36 (90%)	4 (12%)
cg	10 (63%)	11 (69%)	9 (64%)
dc	27 (87%)	33 (92%)	35 (80%)
ep	0 (0%)	3 (75%)	0 (0%)
ft	11 (69%)	17 (94%)	5 (31%)
is	5 (63%)	4 (44%)	5 (45%)
lu	32 (59%)	51 (93%)	9 (21%)
mg	18 (60%)	23 (82%)	14 (38%)
sp	60 (60%)	76 (78%)	18 (20%)
ua	46 (49%)	65 (65%)	32 (34%)

あってもコンパイラの実行が異なれば検出される依存関係の数も異なることが確認できる。このような差異の背景には、ループを命令セットに定義される命令を用いてどのように実現するかという点での実装が依存関係の数に大きく関係していると考えられる。

ループに関する依存関係には、ループ制御に関する操作に関連するループ変数の取り扱いをどのように実現するかはコンパイラや命令セットによる実装が非常に大きく現れると考えられる。例えば、IA64 命令セット定数ループやソフトフェアパイプライニングのための命令と専用のレジスタが定義されている。このようなループに関する命令を用いるかどうかによりメモリアクセスを解する依存の数は変化する。ia64.gcc の環境ではループ命令を多用したコードを生成する傾向があるため検出される依存関係が少ないということにつながったと考えられる。ia64.icc の環境ではループであるならばソフトウェアパイプライニングの命令を用いて命令レベル並列性を得ようとする傾向があるため[10]、検出されるデータ依存数は多くなると考えられる。

4.3 並列ループの推定

並列なループ領域の検出を行うためには抽出したループ領域単位でのデータ依存関係を解析し並列領域を見つけ出す必要がある。この並列領域の探索の手法の一例として lastWriteRegion ポリシを用いて抽出した依存関係からそのループ領域自身への依存の有無を調べるということを行った。lastWriteRegion ポリシを用いて抽出した依存関係においてそのループ領域自身への依存が一度も無いループ領域は、そのループの反復間においてデータ依存関係が無いため潜在的に DOALL ループと考えることが出来る。

表 4 に潜在的に DOALL ループと考えられるそのループ領域自身への依存が無いループ領域の個数を調べた結果を示す。表中の括弧内のパーセンテージはホットループとして検出したループに対する潜在的に DOALL として検出されたループの割合である。結果より、潜在的に DOALL ループであるループがアプリケーション内に存在しそれらを検出できることが確認できる。しかしながら、各環境において潜在的な DOALL として検出されるループの割合は大きく異なることがわかる。特に、潜在的に DOALL と検出された数は IA64 において gcc を用いたとき多い傾向があり、IA64 において icc を用いたときは少ない傾向があることがわかった。このことから、ループの処理の実現方法は命令セットやコンパイラの実装にも密接に関わること見受けられる。

本論文の実行時プロファイリングによる手法では潜在的に DOALL として検出されたループを完全に DOALL ループと断定することは以下の 2 つの理由から困難である。1 つはメモリアクセスを解したデータ依存関係しか確認していないため、レジスタを解したループ反復間の依存がある可能性もあるという理由が挙げられる。もう 1 つは、今回入力に用いたデータセットにおいてたまたま依存が無いことを確認したのみであり、ソースコードの持つ意味的に完全に依存が無いと判断したわけではないため反例が存在する可能性があるという理由が挙げられる。しかしながら、完全な DOALL ループを検出できなくても投機的実行などの機構を利用することによって並列化のために活用することが可能であり、本手法のようにユーザーが並列性について明示することなく並列ループを推定できる本手法は有用であるといえる。

5. 関 連 研 究

Praun らはプログラム中の並列処理の可能性を探るためにデータ依存の観点からメモリアクセスの解析を行う手法を提案している[3]。Praun らのアプローチは我々の提案するデータ依存解析と非常に近い。しかしながら、彼らのデータ依存解析においてはデータ依存解析のための興味領域の指定をユーザーが行わなければならないという制約がある。我々の提案する手法においてはユーザーの明示的なマーカーが無くともループ階層構造を検出し、データ依存解析を行うことが出来る。結果的に、我々の手法はプログラムの動的な挙動についての深い理解がなくてもデータ依存解析を行うことが出来るため有用である。また、Praun らはデータ依存解析により DOALL ループの検出を目指すのではなく、依存の頻度を求め、その逆数として並列性の見積もりを行っている点が本論文とは異なる。

6. 結　　論

本論文では、命令レベル並列性よりも粗粒度なループレベルの並列性をバイナリコード実

行時に抽出することを目的に実行時プロファイリング技術を用いてループ領域毎のデータ依存関係を調べる手法を提案した。提案する手法は、参照した全てのメモリアドレスについて依存関係を把握するために必要な情報を効率的に記録するために、ループ階層構造を利用し、メモリアクセスを解したデータ依存関係をループ領域単位で管理する。

JIT コンパイラ技術を用いた実行時プロファイリングツールを利用して評価環境を構築し、基礎的評価を行った。評価実験の結果より、ループ領域に着目したデータ依存解析は膨大なメモリアクセスの情報を効率的に管理できるということを示した。さらに、抽出したループ依存に着目したデータ依存関係より潜在的な DOALL ループを検出できることを確認した。また、抽出される潜在的な DOALL ループは利用するコンパイラや命令セットの構成と密接に関係があることを示した。

参 考 文 献

1) Michael Chen and Kunle Olukotun. TEST: A tracer for extracting speculative threads. In *Proceedings of the international symposium on Code generation and optimization*, pp. 301–312, 2003.
2) Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: A TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 158–167, 2006.
3) Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 185–196, 2008.
4) Chi-Keung Luk, etal. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pp. 190–200, 2005.
5) Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative optimizations. *ACM Trans. Archit. Code Optim.*, Vol.1, No.3, pp. 247–271, 2004.
6) T. Chen, J.Lin, W.Hsu, and P.C. Yew. Data dependence profiling for speculative optimization. In *In The Proceedings of the 14th International Conference on Compiler Construction*, pp. 57–62, 2004.
7) Yukinori Sato, Ken-ichi Suzuki, and Tadao Nakamura. Run-time detection mechanism of nested call-loop structure to monitor the actual execution of codes. In *Proceedings of First International Workshop on Software Technologies for Future Dependable Distributed Systems*, pp. 184–188, 2009.
8) Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *Proceedings of the 34th annual international symposium on Computer architecture*, pp. 278–289, 2007.
9) JohnL. Hennessy and DavidA. Patterson. *Computer Architecture: A Quantitative Approach 4th edition*. Morgan Kaufmann Publishers Inc., 2006.
10) Sebastian Winkel, etal. Latency-tolerant software pipelining in a production compiler. In *Proceedings of the 6th annual international symposium on Code generation and optimization*, pp. 104–113, 2008.