Vol.2011-ARC-196 No.13
2011/7/27

Loop-Call Context Tree を用いたランタイムデータフロー解析

佐藤幸紀[†1]　井口寧[†1]　中村維男[†2]

超並列マルチコア CPU やさらにアクセラレータを組み合わせた高度な並列処理に代表されるように、発熱やエネルギー浪費の原因となるプロセッサの動作周波数を増加させることなく処理性能を向上させる技術が急速に普及しつつある。しかしながら、現状では並列処理を効率的に行うためにはプログラマがアプリケーション全体の構造を正確に理解し効率的にハードウェアリソースに合わせて並列化することが必要であり、非常にコストのかかる経験的な作業であるといえる。さらに、アプリケーションプログラムは年々その規模と複雑さを増してきているため、大規模・複雑化するアプリケーションプログラムを生産的かつ効率的に並列化する手法を確立することが求められている。本稿ではループ階層構造に着目し大規模・複雑化するアプリケーションプログラムの生産的な並列化の支援を目的として、実行時にループ階層単位のデータ依存関係を示すデータフローを Loop-Call Context Tree を用いてモニターする機構を提案する。ループはプログラム中に様々な粒度で出現する構造であり、年々大規模・複雑化するアプリケーションにおいてもプログラムを適確かつ効率的に並列処理するための重要な手掛かりである。そこで、本研究においては、コンパイル後のバイナリコードからプログラム中に現れるループ構造を抽出すると同時にメモリを介したデータ依存をプログラムの実行時に抽出する機構を動的バイナリトランスレーション技術を用いて構築する。評価環境を構築し評価を行った結果、バイナリコードを入力として実行時にループ階層構造およびループ領域毎のメモリを介したデータ依存関係を抽出可能であることを確認した。

Runtime data flow analysis using Loop-Call Context Tree

Yukinori Sato,[†1] Yasushi Inoguchi[†1]
and Tadao Nakamura[†2]

Advances in parallel processing techniques such as a combination of massively parallelized multicore CPUs and accelerators make it possible to keep performance improvements. However, programmers must fully understand all of program structures across an application in order to perform effective parallel processing, and these said to be emprical and time-consuming processes for them. Additionally, since current application programs become large and complex year by year, we need to realize a productive and effective mechanism that can parallelize large and complex application programs. In this paper, we present a mechanism that monitors runtime data flow using Loop-Call Context Tree. Here, we focus on loop nesting structures which are one of the primary hints for finding parallelism. Using pre-compiled binary codes as an input, we extract loop nesting structures, and at the same time monitor data dependencies through memory access using dynamic binary translation system. We implement our mechanism and evaluated it. From the results, we confirm that we can extract loop nest structures and data depencencies among loop regions.

1. はじめに

CPU1 チップ上に多数のコアを集積するマルチコア CPU の普及やさらに近年注目されている SIMD、FPGA といったアクセラレータを組み合わせたヘテロジーニアスな処理要素を組み合わせた大規模並列システムの開発が急速に進んでいる。これらのような高並列なハードウェアを効果的に利用するためには現在のシステムソフトウェアスタックに加えて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機構を含む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。

高度な並列化の困難さに加えて、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る。特にペタ

†1 北陸先端科学技術大学院大学 情報社会基盤研究センター
Research Center for Advanced Computing Infrastructure, Japan Advanced Institute of Science and Technology
†2 慶應義塾大学
Keio University

Vol.2011-ARC-196 No.13
2011/7/27

スケール、そしてポストペタスケール時代の HPC システムにおいては、多くのアプリケーションはマルチスケール・マルチフィジックス現象の統合シミュレーションという形になるといわれている。このマルチスケール・マルチフィジックスによるアプローチでは、1 つの分野の理論では対応できない複雑な現象を解析するためにミクロからマクロまでの幅広いスケールにわたり様々な物理現象の基礎方程式を解き、現象を再現しようと取り組む。すなわち、アプリケーション内部には特定の現象の解析のため細分化されたモデルや支配方程式、アルゴリズムが多数存在し、それらが複雑なネットワークにより統合された形で単体のアプリケーションとなる。従って、細分化されたレベル毎に並列化するべき対象を的確に見つけ出すことに加えて、アプリケーション全体のデータの流れや並列化対象部分を処理の特性やロードバランスの観点から適切なハードウェア資源にマッピングしていくことが必要となる[1)]。

しかしながら、ハードウェア構成はメモリ階層やアクセラレータ、相互接続方式の面で多様化を続ける一方で、それらを的確にマッピングしていくことは並列アプリケーション開発者にとっては非常に大きな負担となっている。さらに、年を追うごとに規模や複雑さを増大するコードから並列化するべき対象を的確に見つけ出すことも多くの労力を必要とする。従って、大規模な並列性を把握する方法論を確立し高度な並列化を生産的に達成する手法を確立することが必須であると考えられる。

そこで、本研究においてはプログラマがソースコード全体を一通り読まなくともプログラム実行時におけるデータ依存関係やデータの流れのイメージが把握可能となることを目指して、プログラム中のループ階層構造に着目し、ループ構造を基準とするデータ依存をプログラムの実行時にモニターする手法を提案する。

プログラムの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は効果的に並列化対象部分を抽出することに寄与する一方で、参照した全てのメモリアドレスについての膨大な依存関係を的確に記録する必要がある。すなわち単に全てのメモリ参照の履歴を保持するのではなく、データ依存関係を並列部分抽出のために理解しやすい形式として必要十分な情報量で効率的に記録する必要がある。

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、検出したループ領域単位でデータ依存の有無を管理することを試みる[2)]。ループ構造はプログラム中に様々な粒度で出現する構造であり、年々大規模・複雑化するアプリケーションにおいてもプログラムを適確かつ効率的に並列処理するための重要な手掛かりである[3)]。本稿では、文献[4)] において提案されている正確なループ階層構造の抽出法と On-the-fly による Loop-Call Context Tree の生成法をループ階層構造単位のデータ依存解析と組み合わせることにより、Loop-Call Context Tree グラフ上にデータフローグラフを生成する機構を動的バイナリトランスレーション技術を用いて構築する。

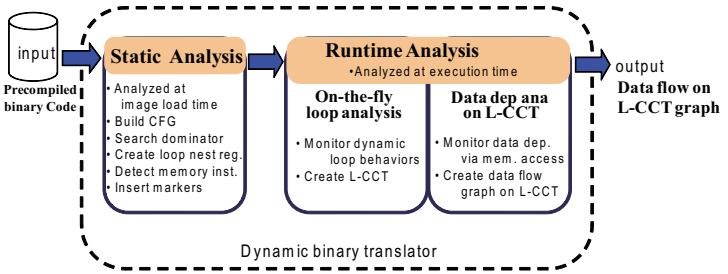

図 1 DBT を用いたにデータフロー解析機構

図 1 に本研究で提案する動的バイナリトランスレーション（DBT）技術を用いたデータフロー解析システムの概要を示す。動的バイナリトランスレーション（DBT）技術は、ランタイム最適化・並列化の実現など新しいマイクロアーキテクチャを確立するうえで重要な技術であると同時に、ハードウェアのレイアに近い情報のプロファイルやハードウェアの動的な挙動のモニタとしても応用されている。本報告では、その応用の一例として DBT により実装したループ階層構造の抽出機構およびメモリを介したデータ依存抽出機構を用いて Loop-Call Context Tree グラフ上にデータフローグラフを生成することを取り上げる。特に、バイナリコードを入力として実行時にループ階層構造およびループ領域毎のメモリを介したデータ依存関係を抽出可能であることをベンチマークプログラムの実行を通して示す。

本論文の構成は以下の通りである。2 節はで DBT 技術によりループ階層構造を抽出する手法の概要を述べる。3 節ではループ領域毎のメモリを介したデータ依存関係を抽出する手法について述べる。4 節ではベンチマークプログラムの実行を通して本手法の基礎的な評価を行う。5 節は結論である。

2. ループ階層構造の抽出

本節では文献[4)] にて報告されたループ階層構造抽出機構である pMarker 法についての概要を説明する。pMarker 法は関数単位にてコントロールフローグラフおよび支配木を構築

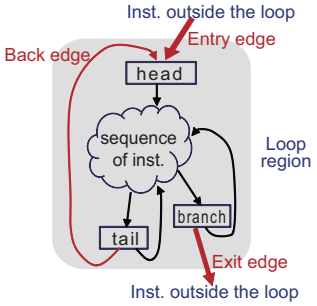

図 2 ループの構造

し Havlak のアルゴリズム[5] により reducible ループと irreducible ループの双方を検出した後、実行時にどのループ領域の内部にいるかを追跡するためのマーカーを生成し、DBT システム上の動的解析における instrumentation のポイントとし、動的なループの挙動を得る。本手法においてはどのように複雑に入り組むループネストを追跡するかが動作の速度や効率を決める重要な要素となる。

図 2 に抽象化した典型的なループの構造を示す。ループは head となる命令、tail となる命令、その他の命令にて構成される。ループの head 命令がループ領域内のすべての命令を支配する場合は reducible ループであり、ループ領域外からの流入は head 以外にない。また、ループ内には少なくとも 1 つの head に戻るエッジがあり、制御フローをループさせる。irreducible ループについては少なくとも 1 つ以上の head 命令以外への流入がある点が異なる。

実行時にループ階層構造を追跡するという目的を実現するためには、すべてのコントロールフローに着目する必要はなく、ループ領域外からの流入とループ領域外への流出に着目すればよい。そこで、pMarker 法ではループへの流入および流出となるエッジを監視することにより、実行時にループ階層構造を効率的に追跡する。なお、文献[2] で用いたループ検出機構は文献[4] における TB 法に相当するため、ループ階層構造を正確に抽出することができない。

pMarker 法は図 1 の静的解析（Static Analysis）とランタイム解析（Runtime Analysis）の組み合わせにより実現される。以下、ループ構造を検出する手法の概要を順を追って DBT システムでの実装に即して説明する。

事前にコンパイルされているアプリケーションプログラムのバイナリコードを入力とし

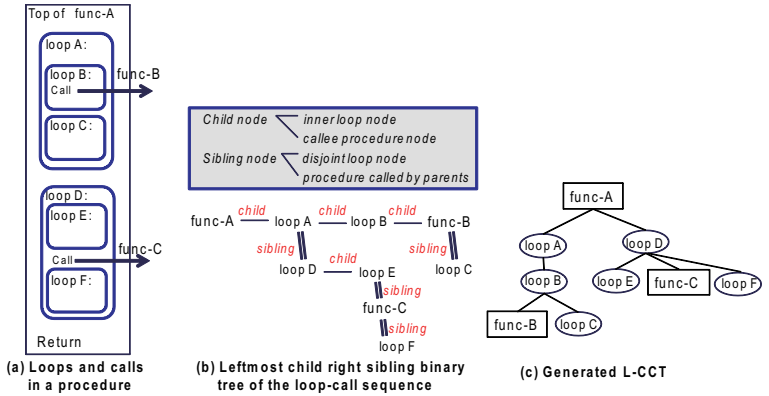

図 3 L-CCT (Loop-Call Context Tree) グラフとデータ構造

て第一のステップとして静的解析を行う。静的解析はバイナリコードのイメージが DBT にロードされるときに行う。静的解析フェーズにおいてはコントロールフローグラフおよび支配木を構築し Havlak のアルゴリズムにより reducible ループと irreducible ループの双方を検出する[5]。加えて、検出したループの領域を示すマーカーと関数呼び出しやリターンの位置を示すマーカーを生成し、動的解析における instrumentation のポイントとする。プログラムの実行を伴わない静的解析においては関数をまたぐ（inter-procedural な）解析や動的なコントロールフローの推定は非常に難しい。そこで、これらをプログラムの実行時に抽出する動的解析を行う。

動的なランタイム解析のステップは以下のように動作する。ランタイム解析では生成したマーカーのポイントが実行される毎にループ情報を記録するための解析プログラムが実行される。ランタイム解析においてはプログラム実行中の条件分岐の挙動やコントロールフローを反映した解析が可能となるため、動的な実行で実際に現れるループ構造に関する情報を抽出することができる。

関数をまたぐプログラム全体のループ階層構造を効率的に保持するために **L-CCT (Loop-Call Context Tree)** というデータ構造を構築する。L-CCT はコールコンテキストプロファイリング[6] にて利用される CCT (Call Context Tree) を拡張し、関数をまたぐループネスト構造を表現できるようにしたものである。

図 3 (a) に関数 func-A のループ階層構造および call 命令の呼び出し位置を示す。関数

func-A は内部に 6 つのループを持ち、ループ B の内部で関数 func-B を、ループ D の内部で関数 func-C の呼び出しを行っている。L-CCT はこれらの呼び出しシークエンスを leftmost child right sibling binary tree（長男次男表現）を用いてにて正確に把握する。図 3 (b) に leftmost child right sibling binary tree 形式にて表現された関数 func-A のループと関数呼び出しの関係を示す。図 3 (c) は最終的に生成される L-CCT を示す。ここで円形のノードはループを、四角のノードは関数を示す。Havlak のアルゴリズムにより検出される 2 つの任意のループはそれぞれがネストしているか、互いに素であるかのどちらかとなるため、ループ内部で呼ばれるノードはそのノードの子ノードとして追加する。すなわち、ネストしているループにおいてはアウターループが親に、インナーループが子になるようにノードが追加される。互いに素の場合はそれぞれが兄弟となるようにノードを追加する。このような L-CCT によりプログラム実行時に実際に実行されたループネスト構造と関数の位置関係を把握することが可能となる。

3. メモリを介したデータ依存の抽出

本節では本研究で提案する動的バイナリトランスレーション（DBT）技術を用いたデータフロー解析システムについて説明する。図 1 に示すようにデータフロー解析システムはループ階層構造抽出機構と連携しつつ静的解析（Static Analysis）とランタイム解析（Runtime Analysis）の組み合わせにより実現される。以下、メモリを介したデータ依存を抽出する手法を順を追って説明する。

第一のステップである静的解析においてはループ構造解析に加えてメモリアクセス命令を検出しメモリアクセス命令に対してマーカーを生成することにより DBT システム上の動的解析における instrumentation のポイントとする。第二のステップであるランタイム解析では、生成したマーカーのポイントが実行される毎にメモリアクセスに関する情報を記録するための解析プログラムが実行され、動的なメモリ依存の挙動を得る。

メモリアクセスの際に実行される解析プログラムにおいては、データ依存関係をループ階層構造に基づき表現する機構を用意する。図 4 にメモリライトおよびメモリリードアクセスの際の動作の概要を示す。メモリを介したデータ依存関係を把握するためにメモリアドレス毎に最後に書き込みを行ったループの ID を保持するテーブルである Last write table を用いる。メモリライトアクセスが実行されるポイントで書き込みを行うメモリアドレスに対応するインデックスの書き込みを行ったループの ID（last Write LoopID）を現在実行されているループ ID に更新する。

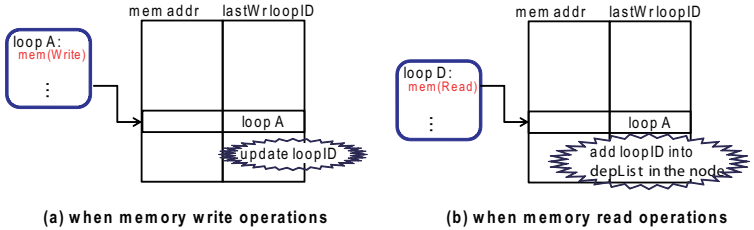

図 4 ループの構造

メモリリードアクセスが実行されるポイントでは読み込みを行うメモリアドレスに対応するインデックスのループ ID を得る。このときのループ ID はそのメモリにある値が生成された領域を示す。すなわち、現在実行されているループはそのメモリにある値が生成された領域と依存しているということになる。このようにして得られた依存関係は 2 節で示す兄弟次男表現による L-CCT 上のノード毎にどの領域と依存しているかを示す depList というリストにて保持される。そこで、現在のループノードに対応する depList に検出された依存のあるループ ID を追加する。

現在実行しているノードが L-CCT 上の関数ノードである場合はループ ID の代わりに関数に対応する ID を用いることにより L-CCT のどのノードにて生成された値と依存関係があるかを把握できるようにする。以上のような動作によりメモリを介したデータ依存をループ領域単位で抽出することが可能となる。

4. 評価実験

4.1 実験環境

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動的バイナリトランスレーション技術を用いて実環境を構築し、ベンチマークプログラムにより基礎的な評価を行う。構築には DBT システムとして Pin tool set[7] を用いた。Pin はよく知られた DBT システムの 1 つであり、同一の ISA への変換を行う。DBT システムにおいては一度変換された命令群はコードキャッシュに保持されるため高速に参照することが可能である。

システムの評価には汎用的な x86 クラスタの 1 ノードを用いた。評価に用いたクラスタの詳細は以下である。ハードウェアとして 4 基の 6 コアを持つ 2.6GHz の CPU (AMD Opteron 8435) と 128GB の主記憶メモリを備える Appro 1143H というサーバーおよび

OS として Red Hat Enterprise Linux 5.4 にて構成される汎用的な x86_64 のクラスタであり、1 ノードあたり 24 コアの CPU が利用できる。この上に Pin のバージョン 2.8 (33586) Intel64 用の構成にて DBT システムの環境を構築した。

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 の逐次版の IS および CG を利用した。データサイズは class A を用いた。IS は C 言語により実装される大規模整数ソートであり、CG は共役勾配法のカーネルである。コンパイラとして gcc4.1.2 に '-O' オプションを用いてバイナリコードを生成した。

ループネスト解析に関しては、実行バイナリに含まれる領域のみを解析の対象として、動的にリンクされるライブラリは解析の対象から外した。加えて、解析は main 関数が実行される時点から開始し main 関数が終了した時点で停止することとした。また、プログラム全体の L-CCT は非常に大きくなるため、実行頻度が高い Hot 領域を興味対象とした Hot L-CCT を結果の解析に用いた。ここで Hot ノードを全命令実行数の 1%以上を占めるノードとして、Hot 領域は Hot ノードおよび Hot ノードを子孫に持つノードと定義した。Hot L-CCT の可視化は graphviz により行った。

4.2 評価結果

図 5 に is.A の Hot L-CCT 上にメモリを介したデータ依存関係を追加したデータフローを、図 6 に同じく cg.A のデータフローを示す。丸いノードはループを示し、四角のノードは関数を表す。実線のエッジは L-CCT グラフを表し、点線のエッジはデータ依存関係があることを示す。本結果においてはデータ依存関係のうち自分自身のノードと依存関係がある場合はエッジは追加しないとした。

データ依存関係に加えてプログラム実行の挙動を把握しやすくするため各ノードの統計的な情報をノード内部に示した。ループノードの場合にはループ ID、関数ノードの場合には関数名が上段に示され、下段にはプログラム全体におけるそのノードの実行命令数の割合、および括弧内にそのノードの子孫まで含めた実行命令の割合が示されている。また、ノードの右下にはループトリップカウントの平均値、およびそのループが出現し実行された回数（ループ外からそのループに新しく流入した回数）を示す。ループトリップカウントは各ループの反復回数の総和をループ出現回数で割った値である。

結果より、ループ領域毎のメモリを介したデータ依存関係の概略が抽出可能ということが分かる。しかしながら、Hot L-CCT 上に示されるデータ依存関係だけではノードの独立性を完全に示しているといえないことに注意しなければならない。例えば、Hot L-CCT に含まれるノードであってもデータ依存のエッジを持たないノードがいくつか見られた。しかし

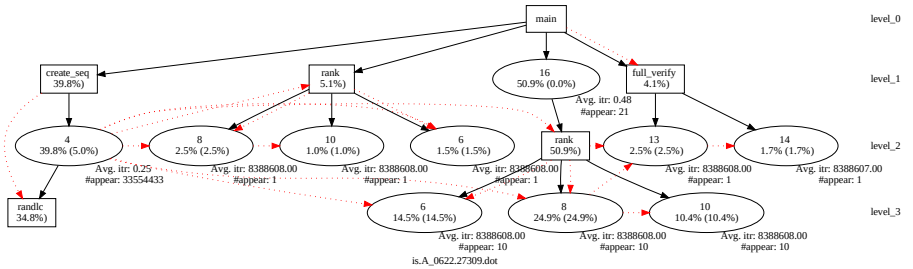

図 5 is.A の Hot L-CCT におけるデータフロー

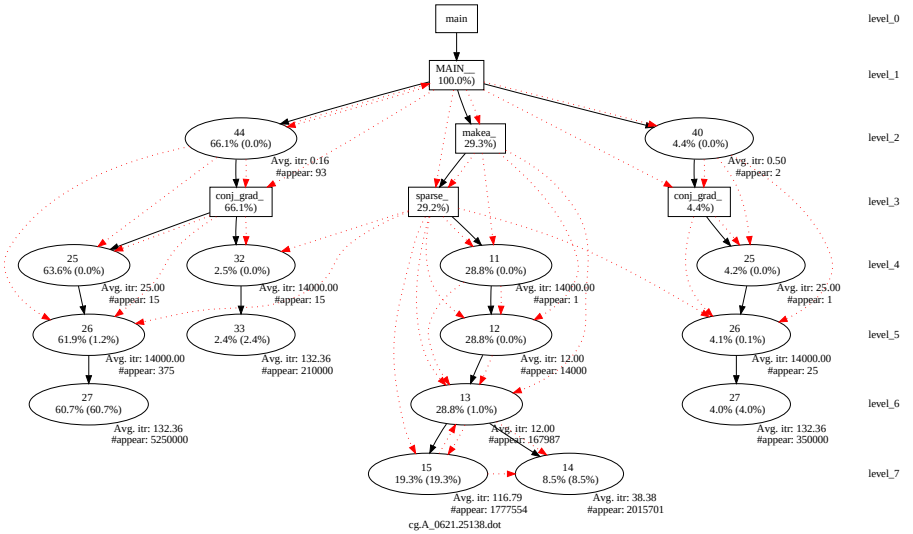

図 6 cg.A の Hot L-CCT におけるデータフロー

ながら、Hot loop に含まれないノードからのデータ依存、レジスタを介したデータ依存、自分自身のノードへの依存が本グラフから省略されていることを踏まえて考える必要があり、必ずしもデータ依存のエッジを持たないノードが他のノードから独立ということではないことに注意する必要がある。

本手法で抽出されるデータフローは L-CCT に含まれるすべてのノードに対してのデータ依存関係である。しかしながら、L-CCT に含まれるすべてのノードに対してデータ依存関

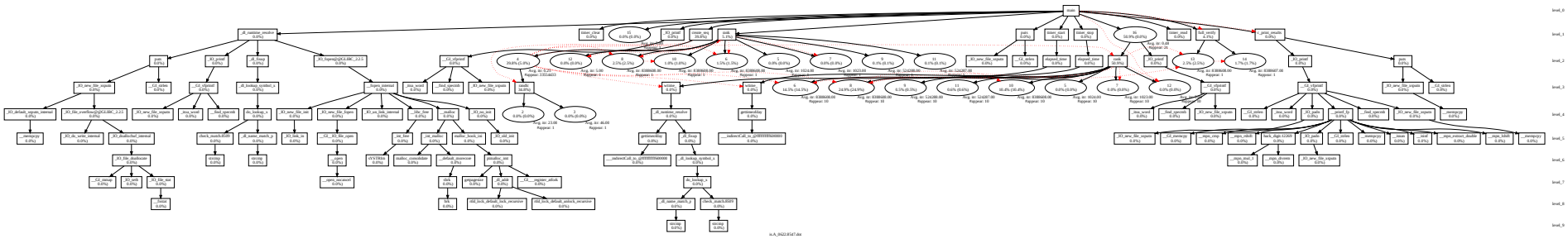

図 7 is.A の L-CCT におけるデータフロー

係を示すことは、実行中に呼び出されたすべてのノードを表示することになるので、可視化により人間が理解することを支援するかという面では必ずしも適切でないといえる。図 7 に is.A の L-CCT に含まれるすべてのノードを対象としたメモリを介したデータフローを示す。すべてのノードを対象とした場合、ノード間のデータ依存関係は適切に抽出されることとなるが、ノード数が膨大となるためにプログラムの挙動が直感的に理解することができない。今後の課題として、並列性抽出を支援するために必要十分の情報を表す手法を確立していくことが挙げられる。

5. 結　　論

本稿では、大規模・複雑化するアプリケーションプログラムの生産的かつ効率的な並列化の支援を目的としたデータフローをモニタする機構、およびその可視化手法を提案した。本手法においては、ループ階層構造をプログラムを適確かつ効率的に並列処理するための重要な手掛かりとして着目し、コンパイル後のバイナリコードからプログラム中に現れるループ構造を抽出すると同時にメモリを介したデータ依存をプログラムの実行時に抽出する。本機構を動的バイナリトランスレーションシステム上に構築し評価を行った。その結果、バイナリコードを入力として実行時にループ階層構造およびループ領域毎のメモリを介したデータ依存関係を抽出可能であることを確認した。

参　考　文　献

1) 佐藤幸紀. ループ構造に着目したマルチグレイン・マルチレイヤ並列処理システムの提案. 情報処理学会研究会報告 2008-ARC-172, pp. 25–28, 2008.
2) 佐藤幸紀, 中村維男. 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. 情報処理学会研究会報告 Vol.2009-ARC-184 No.8, pp. 1–8, 2009. 2009 年 並列/分散/協調処理に関する『仙台』サマー・ワークショップ SWoPP2009.
3) 佐藤幸紀, 井口寧, 中村維男. 動的バイナリトランスレーションによるループネスト検出とプログラムチューニング支援への応用. 情報処理学会研究会報告 Vol.2010-ARC-192 Vol.2010-HPC-128 No.10, pp. 1–7, 2010. 第 18 回ハイパフォーマンスコンピューティングとアーキテクチャの評価に関する北海道ワークショップ（HOKKE-18）.
4) Yukinori Sato, Yasushi Inoguchi, and Tadao Nakamura. On-the-fly detection of precise loop nests across procedures on a dynamic binary translation system. In *2011 ACM International Conference on Computing Frontiers*, 2011.
5) Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, Vol.19, No.4, pp. 557–567, 1997.
6) Glenn Ammons, Thomas Ball, and JamesR. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pp. 85–96, 1997.
7) 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.