Vol.2013-HPC-140 No.3
2013/7/31

バイナリ変換による透過的なループ構造解析とコード実行時の区間実行時間の計測

佐藤 幸紀[1,2,a)]

概要：ループ階層構造はマルチグレイン・マルチレイヤにまたがる並列性を理解する重要な手掛かりであり、ループ階層を意識した並列性はマルチコア環境下の超並列処理の性能チューニングにおいて鍵となる技術である。しかしながら、現状ではアプリケーションに由来する多種多様な粒度の並列度を自動で解析するツールはなく、高度な知識を持つプログラマによる手作業の解析とチューニングに依存している状況である。そこで、並列化の戦略を立てる上で有用となるループ階層構造に着目した区間実行時間のプロファイリングを行う機構を提案し、バイナリ変換技術を利用してプログラマから透過的にループ領域単位の実行時間を実行時に計測する。本機構により自動かつ生産的に実行時間をプロファイルすることによりチューニングするべき区間の特定の支援を行う。加えて、基礎的な評価を行った結果から、解析オーバーヘッドと測定対象のアプリケーションのキャッシュの挙動との関連について考察する。

1. はじめに

エクサスケール時代に予想される HPC アプリケーションの方向性として、シミュレーションのマルチスケール・マルチフィジクス化が挙げられる。マルチスケール・マルチフィジクスシミュレーションにおいては、多様な物理現象をモデル化するために必要となるコード規模とその複雑性が年々増加の一途をたどっている。また、近年、実社会の課題をセンシング技術や情報処理技術の組み合わせにより解くことを試みるサイバーフィジカルシステムが注目を集めている。このような時代のアプリケーションの要求としては、マルチスケール・マルチフィジクスシミュレーションの更なる多機能化や大規模グラフ探索への対応などが求められている。

アプリケーションプログラムの高度化に伴い、他人の書いたコードの利用やライブラリを利用する機会の増加することによりコード可読性の悪化する一方で、複数のカーネルが処理のホットスポットとして実行時間に影響を与えるようになり、多様で多階層の並列性をハンドリングする膨大な量のソースコードから多階層の並列性を抽出し適切なハードウェアにマッピングするという高度な並列化や性能チューニングを生産的に達成することが必須となりつつある。

高性能計算機システムのトレンドとしては、CPU のマルチコア・メニーコア化が急速に進み計算ノード内においても高並列な処理を行う必要性が増していることに加えて、単一システム構成においては高速ネットワークを用いて計算ノードを大規模に並列処理ノードとして展開することから計算ノードレベルの並列度も増加の一途をたどっている。更に、アクセラレータを用いた演算の高速化、メモリスループットを向上させるための新しいデバイスの利用やそれらに伴うメモリの階層化が急速に進んでいる。このようなシステム構成において効率的に大規模並列処理を行うためには、多階層の並列性を意識しつつ超並列な処理を円滑に進める必要がある。

多階層の並列性を取り扱うことにおいてループ階層構造はマルチグレイン・マルチレイヤにまたがる並列性を理解する重要な手掛かりであり、ループ階層を意識した並列性はマルチコア環境下の超並列処理の性能チューニングにおいて鍵となる技術である。しかしながら、現状ではアプリケーションに由来する多種多様な粒度の並列度を自動で解析するツールはなく、プログラマによる人力のチューニングに依存している状況であり生産性が低く問題となっている。

図 1 に多階層にわたる並列性とハードウェアへのマッピング手法の対応関係を示す。並列性の階層は、命令レベル並列性、粗粒度ループ並列性、プロセスレベル並列性に分類できる。命令レベル並列性の領域は、最内ループを対象とし、伝統的なコンパイラでの命令スケジューリングや HW におけるアウトオブオーダ実行により並列化される。

[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
[2] JST CREST
[a)] yukinori@jaist.ac.jp

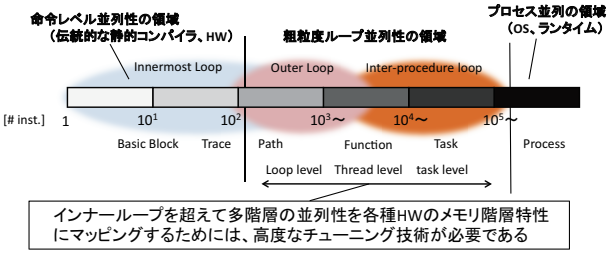

図 1 ループ階層構造

プロセスレベル並列性の領域においてはプログラマに明示的に記述された並列区間の並列実行が OS やランタイムのサポートにより行われる。しかしながら、粗粒度並列ループの領域においては、アウターループ、関数を跨ぐループを手動で抽出し、ループレベル並列性やスレッドレベル並列性、タスクレベル並列性として対象とする HW の並列性やキャッシュメモリ特性にマッピングするためには、高度なチューニング技術が必要である。

このような状況の下、ループ階層構造に着目し並列化の戦略を立てる上で必要となる情報のプロファイリングを行うツールとして実行駆動型アプリケーション解析ツールを開発してきた [1] [2] [3]。これらは処理のホットスポットの発見と、生産的な並列化を達成することを目指したツールであり、ループ階層構造を単位とした解析に特に着目している。これらのツールはプログラム全体をコンパイル済みのバイナリコードを入力としてコード実行時にプロファイリングし、多種多様なループ階層構造やそれらの間の並列性を検出する。本ツールは、動的ライブラリがリンクされているバイナリコードにおいても、動的ライブラリの実行区間も含めて全てのコード実行をプロファイルすることが可能である。また、MPI のコードもプロセス毎にその挙動の解析を行うことができる。加えて、デバッグ情報付加によるソースコードとの対応付けやソースコードのないバイナリコードの解析も可能である。

しかしながら、本ツールを利用したメモリ依存を含むプロファイルの実行には、解析を行わないネーティブのコード実行と比べて SPEC2006CPU の平均で約 50 倍の実行スピードの低下、約 20 倍のメモリ消費が発生することが報告されている [2]。特に、プロファイリング時間のオーバーヘッドは生産的にこれらのプロファイリングを利用する上での大きな制約となる。加えて、チューニングへの応用のためには、ループ階層構造をキャッシュの挙動を如何に結びつけるかが鍵であり、本ツールでキャッシュ性能を推測できるかということが課題である。

そこで本稿では、ループ階層構造を区間として区間実行時間をプロファイリングする機構の実装を行い、並列化の戦略を立てる上で有用となるチューニングするべき区間の特定を支援する。本機構はバイナリ変換技術を利用してプログラマから透過的にループ領域単位の実行時間を実行時に計測する。本ツールの基礎的な評価を姫野ベンチマークおよび物質科学用の計算プログラムである OpenMX により行い、解析オーバーヘッドと測定対象のアプリケーションのキャッシュの挙動との関連についての考察を行う。

2. 実行駆動型アプリケーション解析ツール Exana の概要

多階層にわたる並列性とハードウェアへのマッピング手法を円滑に行い潜在的な粗粒度並列ループをプロファイリングするツールとして実行駆動型アプリケーション解析ツールを開発した。**図 2** に実行駆動型アプリケーション解析ツール　Exana (Execution-driven application analyisis tool) の概要とその出力結果を示す。本ツールは、任意の x86 バイナリコードを入力し、ループ階層構造やそれらの間のデータ依存関係を実行時に解析する。

Transparent Instrumentation フェーズにおいて、バイナリコード中の機械語命令を逆アセンブル後にコンパイラが行っている制御フロー解析と同等の解析を行い、動的解析においてループ、メモリアクセスをモニタするための Instrumentation Code を挿入する。本フェーズにおける処理はバイナリコードのイメージがバイナリ変換システムのコードキャッシュにロードされるときに行われる。

次に、Runtime Analysis フェーズにおいて、解析のための Instrumentation Code が挿入された実行コードを実行し、実行時に出現するループネスト構造と全てのメモリアクセス命令に関するアドレス情報などのランタイムの情報を抽出し記録する。これにより、コンパイル時には困難なポインタによる間接メモリアクセスやコントロールフローも解析可能となる。結果として、ループネスト構造や関数を単位としてそれらの間のデータ依存関係、入力データに依存するループ反復回数 (ループトリップカウント)、関数の出現頻度、プロファイリングに必要な時間を計測することが可能である。

本稿では、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 **CCT (Call Context Tree)**, **LCCT (Loop-Call Context Tree)**, **LCCT+M (Loop-Call Context Tree with Memory dataflow)** というデータ構造を構築し出力するプロファイリングを行う。LCCT はコールコンテキストプロファイリング [4] にて利用される CCT 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り [1] 、LCCT+M は LCCT メモリ依存情報を追加したものである [2]。

図 3 (a) にコールフローグラフを示す。各ノードは関数に対応している。関数 A は関数 main および関数 D より呼ばれているためコールフローのマージがあることがわかる。このマージにより呼び出しシークエンスに由来するネストの親子関係を正確に把握することができない。図 3 (b) に CCT を示す。CCT は呼び出しシークエンスに由来

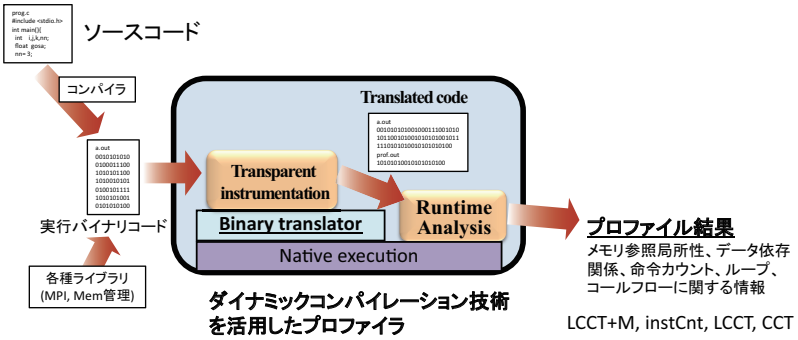

図 2 本論文で用いるプロファイラ機構

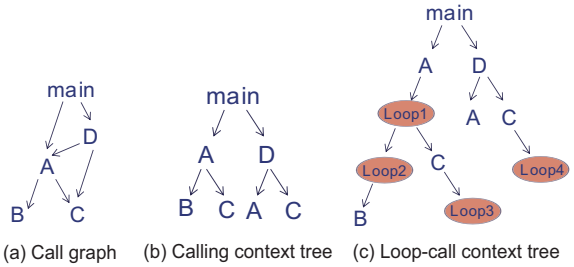

図 3 プロファイリングの手法

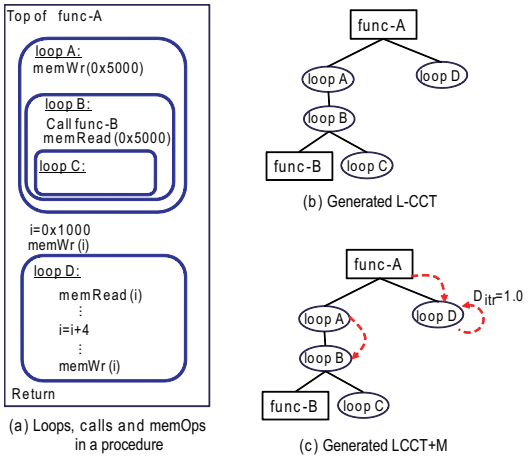

図 4 LCCT+M グラフとデータ構造

するフローセンシティブなパスを表現することが可能である。図 3 (c) は LCCT を示す。LCCT は CCT にループを示すループノードを追加した表記である。Havlak のアルゴリズムにより検出されるループにおいては、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。ネストしている場合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を追加する。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 LCCT 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。

LCCT においてループノード内の数字はループ ID を表している。このループ ID は呼び出しシークエンスに由来して管理されており、ソースコード上で同一な箇所のループであっても呼び出されたコンテクストが異なる場合は異なる ID を持つ。図 3 (c) の例においてはループ 1 は関数 main から関数 A が呼ばれたときのみループが実行され、関数 D から関数 A が呼ばれたときはループが実行されない制御フローをとることが読み取れる。

次に、LCCT+M の詳細とソースコードとデータ構造の対応について述べる。**図 4** (a) は関数 func-A のループ階層構造、call 命令、メモリ命令とそれぞれの相対位置を示す。関数 func-A は内部に 4 つのループを持ち、ループ B の内部で関数 func-B を呼び出しを行っている。LCCT 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 4 (b) に LCCT にて表現された関数呼び出しとループの呼び出しのコンテクストフローを示す。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。ループ内部で呼ばれるインナーループを示すノードはそのノードの子ノードとし、また、素であるループは兄弟ノードとして追加する。図 4 (c) は最終的に生成される LCCT+M を示す。LCCT+M は LCCT に加えてデータ依存のあるノード間に点線で示されるエッジが追加されている形式となる。

3. 区間実行時間のプロファイリング

本稿では、ループ階層構造を区間として区間実行時間をプロファイリングする機構の実装を行い、並列化の戦略を立てる上で有用となるチューニングするべき区間の特定を支援する。本機構はバイナリ変換技術を利用してプログラマから透過的にループ領域単位の実行時間を実行時に計測する。姫野ベンチマークおよび物質科学用の計算プログラムである OpenMX を用いて本ツールの基礎的な評価を行い、解析オーバーヘッドと測定対象のアプリケーションのキャッシュの挙動との関連についての考察を行う。

実行駆動型アプリケーション解析ツールである Exana プロファイラがプログラマの生産的なチューニングを支援するツールとして利用する場合に、HPC コード開発の現場において、どの程度実用的に利用できるか、また、生産

性が向上するかどうかといったユースケースの分析を行った。HPC のコード開発の現場では、ループの最適化はソースコードレベルで取り扱うのが一般的であり、関数内の多重ループに関しては発見するのにそれほどの労力はかからないと考えられている。一方で、例えば、流体と構造の連成解析のようなマルチフィジクスでは、並列化する対象はループだけとは限らずタスクレベルの並列性も考慮する必要があるため、本 Exana プロファイラにより関数とループのコンテクストによりコードを俯瞰することは非常に有益であると考えられる。

課題としては、プロファイリング時間のオーバーヘッドである。データ依存プロファイルは並列性解析には有用であるがプロファイルに必要な時間のオーバヘッドが非常に大きい。データ依存プロファイルをなしとすると解析オーバーヘッドは大幅に小さくなる。従って、ソースレベルでのチューニングに限定した応用であるのであれば、データ依存関係はソースコードを読むことにし、プログラムのコードを俯瞰する手段として特に本ツールを活用するべきとも考えられる。このようなチューニングへの応用のためには、コードの構造に加えてループ階層構造をキャッシュの挙動と如何に結びつけて解析を行うかが鍵であり、本ツールでキャッシュ性能を推測できるかということは検討が必要な項目である。

そこで、ループ階層構造とキャッシュの挙動の関係を把握することを目的としてコード実行およびコード解析における区間タイムを計測する手法の設計および実装を行った。高速な時間計測のためインラインアセンブラにて rdtscp によりマシンクロックサイクルを取得し区間毎にサイクル数の差を求めることにより時間の尺度とした。このインラインアセンブラによる時間計測のためのカウンタへのアクセスをコード解析ルーチンの最初と最後に挿入し、現在実行しているコード解析ルーチンの最初のサイクルカウントから前回実行されたコード解析ルーチンの最後のサイクルカウントの差をとったものを直前に実行した区間のコード実行時間に、現在実行しているコード解析ルーチンの最後のサイクルカウントから最初のサイクルカウントの差をとったものをコード解析ルーチンのための実行時間してそれぞれ求めた。同時に、コード解析部の時間を短縮するために、動的ライブラリが呼ばれて戻るまでの解析を Off とする改良や、基本ブロック単位で計測していた命令の分類毎の命令カウントの取得ルーティンである instCnt の有無を選択できるような改良を行った。

4. 評価実験

4.1 実験環境

提案する区間時間計測機構をを評価するために動的バイナリ変換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。本機構の構築には動的バイナリ変換（DBT) システムとして Pin tool set[5] 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 (Appro gB222X-SM32) の詳細は以下である。クラスタのノード構成はノード毎に 2 基の Xeon X5570 CPU と 24GB の主記憶メモリを備える。OS として Red Hat Enterprise Linux 5.4 が稼働する。本システムは汎用的な x86_64 のクラスタであり、1 ノードあたり 8 コアの CPU が利用できる。この上に Pin のバージョン 2.11 (49303) 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、姫野ベンチマーク [6] と物質科学用の計算プログラムである OpenMX[7] を実アプリケーションプロファイルのケーススタディとして使用した。姫野ベンチマークにおいては、逐次の C static allocation 版のデータセット M を利用し、実行コードは Intel C++ Compiler 11.1 を用いて '-O3 -g ' オプションにて生成した。OpenMX は、密度汎関数理論にもとづく電子状態計算をオーダー N で実行するコードである。ループプロファイルの環境として、OpenMX 入力データには Methane.dat、コード生成にはインテルコンパイラを利用し、-O2 -g オプションと MKL ライブラリを用いて、MPI、OpenMP をオフにした逐次版のコードとした。本コードを用いて、時間とメモリのオーバーヘッドを評価した。

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ループネスト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 LCCT+M 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て結果の解析に用いた。本報告では Hot ノードを決める閾値として子孫を含む累計の実行命令数が全命令実行数に占める割合を用いる。閾値の値はノード自身が実行した命令数が 8 番目に大きいノードの子孫を含む累計命令実行の割合を用いた。また、Hot 領域をグラフとして可視化するためのツールとして graphviz を用いた。

4.2 評価結果

表 1 に姫野ベンチマークの区間実行時間の測定結果を示す。実行時間はプロファイリングを行わない Native と 3 種類のプロファイリング手法と、各プロファイリングにおける命令カウント (instCnt) の有無による結果についてそれぞれ示されている。姫野ベンチマークは測定時間が約 1

表 1 姫野ベンチマークの区間実行時間の測定結果

	Native	with instCnt			w/o instCnt		
		LCCT+M	LCCT	CCT	LCCT+M	LCCT	CCT
MFLOPS	5154	2.5	1348	1685	2.7	2377	4150
Time ratio	1	2062	3.82	3.06	1909	2.17	1.24
App. ratio	100%	3.4%	57.8%	60.8%	3.3%	60.9%	88.5%
Est. app. overhead	-	70.1	2.21	1.86	62.0	1.32	1.10

分になるようにカーネルの反復数が調整されているため、スコアとして提示される MFLOPS の逆数をとることにより時間の尺度 (Time ratio) とした。App. ratio の行にはそれぞれのプロファイリングにおけるアプリケーションコード実行に必要とされた時間の割合を示す。アプリケーション実行部分の割合より算出したアプリケーションの実行のために必要な区間実行時間を Native での実行時間と比べたものが推定アプリケーションオーバーヘッド (Estimated application overhead) である。

結果より、データ依存プロファイルを行うことはプロファイリングのための解析コード部の実行オーバヘッドだけではなく、アプリケーションコード実行時間にも大きなオーバヘッドを与えている事がわかる。例えば、instCnt が有りの LCCT+M は Native 実行の約 2000 倍のオーバヘッドがあることが分かる [*1]。その 96.6%はプロファイラのための解析コードの実行が占める。残る 3.4%の部分がアプリケーションコード実行のための時間であるが、Native の実行時間と比べると 70 倍のオーバヘッドとなっていることが分かる。

instCnt が無の場合は、ありの場合と比べて全てのプロファイリング手法においてオーバーヘどが小さくなっていることが分かる。これは、instCnt は全ての基本ブロックの先頭に解析コードを挿入するためであり、特に、LCCT や CCT のようなプロファイリング手法においては影響が大きいことが分かる。また、LCCT と CCT を比べた場合には、LCCT はループ部分の解析のために全てのループイテレーションの先頭に解析コードを挿入しているため CCT とくらべてオーバヘッドが大きいことが分かる。

最も解析オーバヘッドの小さい CCT で instCnt が無の場合においてもアプリケーションの実行が 1.1 倍の速度低下があることが分かった。このアプリケーションコード実行部の速度低下はプロファイルのためのコードの影響によりキャッシュの挙動が変わってしまったことに由来していると推測される。本アプリケーション解析ツールにて観測しているものは、ループ単位で各領域の実行時間を測定可能である反面、オリジナルのキャッシュの挙動を乱した時間を観測していることになると推測される。キャッシュの挙動を観測するツールとして利用するためにはキャッシュヒット率というのデータと合わせて解析をし、その区間時間計測の観測に由来する実行時間変化についてより詳細に調べる必要である。

図 5 に生成された姫野ベンチマークの LCCT+M を示す。ここで、丸いノードはループを示し、四角のノードは関数を表す。実線のエッジはコントロールフローを表し、点線のエッジはノード間のデータ依存を示す。本グラフにおいて呼び出し元の関数あるいはアウターループが親ノードに、呼び出された関数やインナーループが子ノードとして表現される。各ノード内には自身のノードで実行された命令の全実行命令に占める割合およびカッコ内にそのノードと子孫の総計の実行命令の割合が示されている。すなわち、ノード内の%は計測したアプリケーション実行時間の全体を 100%としたときの各ノードの占める割合を示す。また、各ループノードの内部には、ループを識別するためのループ ID と、バイナリコードの位置に対応するデバッグ情報から得られたソースコード上の位置 (*filename:line*) がそれぞれ示されている。結果より、データ依存関係と共にループ区間や関数がどの程度の実行時間を占めているのかが俯瞰的に理解できることが分かる。

次に、評価コードとして OpenMX を用いた実アプリケーションを用いた評価を行った。**表 2** に OpenMX の区間実行時間の測定結果を示す。実行時間は LCCT、CCT によるプロファイリング手法について命令カウント (instCnt) の有無についてそれぞれと、プロファイリングを行わない Native についての結果が示されている。OpenMX では、単純に Methan.dat を入力としたときの実行時間をそれぞれ示している。Time ratio は Native を 1 としたときの実行時間の割合を示す。また、姫野ベンチマークと同様に App. ratio の実測値より算出した推定アプリケーションオーバーヘッド (Estimated application overhead) を示す。

最も解析オーバヘッドの小さい CCT で instCnt が無の場合においてもアプリケーションの実行が 1.9 倍の速度低下があることが分かった。また、姫野ベンチマークの結果と比較して、このアプリケーションコード実行部の速度低下が若干大きいことが読み取れる。これから、実行カーネルだけを対象とした場合と比べて、実際のアプリケーションレベルの挙動をプロファイリングするためには単純なカーネルのプロファイリングに比べて、より複雑な挙動に

[*1] [2] にて提示されているデータ依存プロファイルの平均的な値よりも大きな値になっているのは、[3] におけるワーキングデータセットの解析のためにハッシュテーブルへのアクセス方法を変えた実装となっているためである。

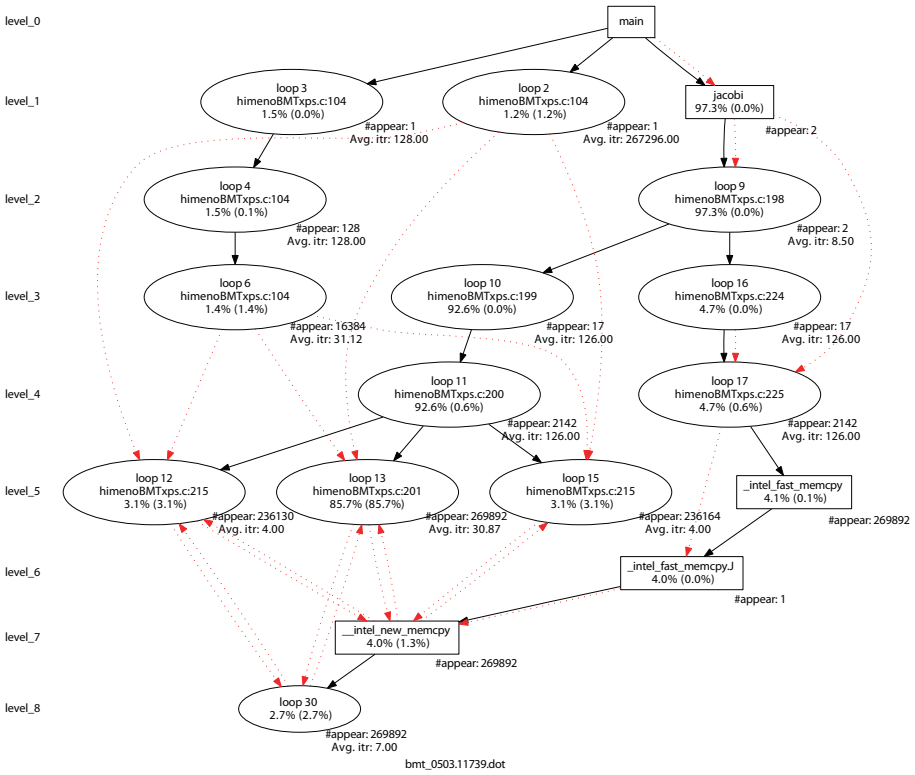

図 5 生成された姫野ベンチマークの LCCT+M

表 2 OpenMX の区間実行時間の測定結果

	Native	with instCnt		w/o instCnt	
		LCCT	CCT	LCCT	CCT
Time [s]	26.2	291.6	249.2	120.5	70.8
Time ratio	1	11.1	9.5	4.6	2.7
App. ratio	1	56.5%	57.8%	58.7%	69.6%
Est. app. overhead	-	6.3	5.5	2.7	1.9

対応するためのデータ構造を保持するためにオリジナルのキャッシュの挙動を乱してしまっていることが推測される。

図 6 に OpenMX における LCCT の結果を示す。本結果の可視化においては、実行される命令が全実行命令数の 2.74%を超える領域を処理のホットスポットとして、その部分のみの表示を行った。結果より、関数とループのコンテクストツリーによりソースコードを読まなくともコード実行の概要を把握することが可能であることがわかる。例えば、main 関数の中で関数 readfile、loop 10、関数 outfile がそれぞれ一度ずつ呼ばれ、それらの以下に深いループネストがあることが観測された。

OpenMX は、269 個の.c ファイルと 22 個の.h ファイルから構成され、それらの累計が 271647 行の規模のアプリケーションである。この様な規模のアプリケーションであっても本ツールを利用することによりソースコードを一度も読まなくともループレベルのホットスポットの位置をある程度の直感的に理解できる事がわかる。LCCT を出力するプロファイリングを行うためには CCT と比べて若干のオーバーヘッドが増えるが、表 2 に示されたように Native 実行の 5 倍程度の速度低下で実際に利用されているアプリケーションを解析できることが分かる。以上のように、データ依存関係と共にループ区間や関数がどの程度の実行時間を占めているのかが俯瞰的に理解できることが分かる一方で、アプリケーションの各区間の実行時間は解析のために挿入されたコードの影響によるキャッシュの挙動の変化に影響を受けている可能性あるため、今後検証が必要である。

5. まとめと今後の課題

ループ階層構造はマルチグレイン・マルチレイヤにまたがる並列性を理解する重要な手掛かりであり、ループ階層を意識した並列性はマルチコア環境下の超並列処理の性能チューニングにおいて鍵となる技術である。しかしながら、現状ではアプリケーションに由来する多種多様な粒度の並列度を自動で解析するツールはなく、高度な知識を持つプログラマによる手作業の解析とチューニングに依存している状況である。そこで、並列化の戦略を立てる上で有用となるループ階層構造に着目した区間実行時間のプロファイ

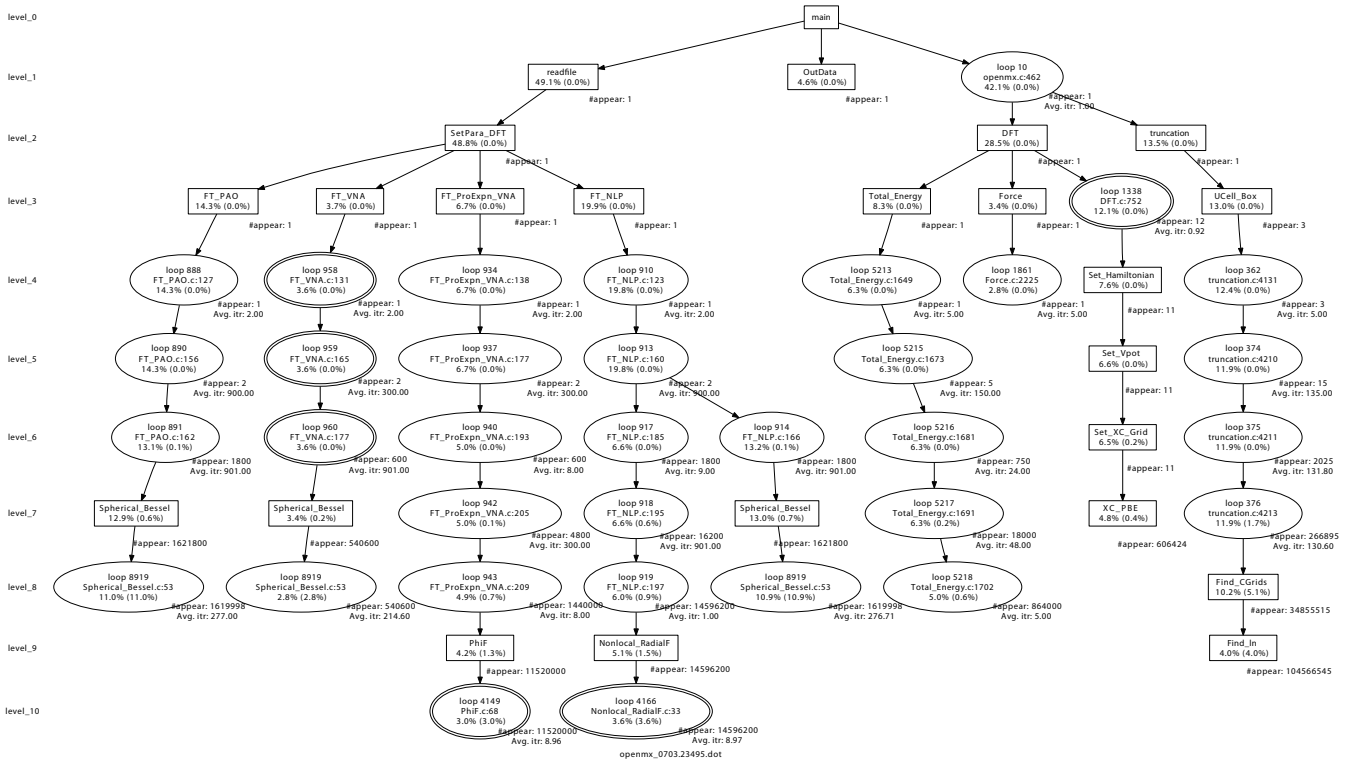

図 6 生成された OpenMX の LCCT

リングを行う機構を提案し、バイナリ変換技術を利用してプログラマから透過的にループ領域単位の実行時間を実行時に計測する。本機構により自動かつ生産的に実行時間をプロファイルすることによりチューニングするべき区間の特定の支援を行う。加えて、基礎的な評価を行った結果から、解析オーバーヘッドと測定対象のアプリケーションのキャッシュの挙動との関連について考察する。

本稿では、生産的なチューニングを支援することを目的に開発してきた実行駆動型アプリケーション解析ツールがHPC コード開発の現場において、どの程度実用的に利用できるか、また、生産性が向上するかどうかといったユースケースを想定し、ツールとして求められる機能を検討した。その結果から、並列化の戦略を立てる上で有用となるループ階層構造に着目した区間実行時間のプロファイリングを行う機構を提案し、バイナリ変換技術を利用してプログラマから透過的にループ領域単位の実行時間を実行時に計測する手法を実装した。評価実権の結果、実アプリケーションでのコード解析においても解析するべき情報を取捨選択することによりオーバーヘッドは削減できることを示した。また、ループ領域の実行時間は観測されるが、キャッシュ性能との連携は観測オーバーヘッドの面で課題があることが分かった。

今後の課題として、ループ階層構造とキャッシュの挙動の関係を調べる機構の開発、解析の興味領域をユーザーが指定するインターフェースの実装、さらに、HW プリフェッチおよび SW プリフェッチとキャッシュ性能との連携などの面でのツールの機能強化が挙げられる。

謝辞

本研究の成果の一部は SS 研マルチコア WG の活動における各委員のメンバーとの議論により生まれたものである [8]。この場を借りて感謝の意を表する。

参考文献

[1] Sato, Y., Inoguchi, Y. and Nakamura, T.: On-the-fly Detection of Precise Loop Nests across Procedures on a Dynamic Binary Translation System, *Proceedings of the 8th ACM International Conference on Computing Frontiers* (2011).

[2] Sato, Y., Inoguchi, Y. and Nakamura, T.: Whole program data dependence profiling to unveil parallel regions in the dynamic execution, *2012 IEEE International Symposium on Workload Characterization (IISWC2012)*, pp. 69–80 (2012).

[3] Sato, Y., Midorikawa, H. and Endo, T.: Identifying Working Data Set of Particular Loop Iterations for Dynamic Performance Tuning, *6th Workshop on Architectural and Microarchitectural Support for Binary Translation (AMAS-BT2013). Held in conjunction with the 40th Int'l Symposium on Computer Architecture (ISCA-40)*, pp. 1 –6 (2013).

[4] Ammons, G., Ball, T. and Larus, J. R.: Exploiting hardware performance counters with flow and context sensitive profiling, *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pp. 85–96 (1997).

[5] Luk, C.-K. et al.: Pin: building customized program analysis tools with dynamic instrumentation, *Proceedings of*

the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 190–200 (2005).
[6] : http://accc.riken.jp/2444.htm.
[7] : http://www.openmx-square.org/.
[8] 佐藤幸紀：実行駆動型アプリケーション解析ツール Exana, サイエンティフィック・システム研究会 マルチコアクラスタ性能 WG 成果報告書 「実践、アプリ高速化に向けて」(2013).