メモリ階層対応ダイナミックコンパイレーション機構の動作原理とコードプロファイリング

佐藤 幸紀[1,2,a)]

概要：ヘテロジーニアスで多階層なメモリアーキテクチャを持つシステムにおいて高い性能効率を得るためには既存のキャッシュメモリやコンパイラでは考慮できないデータ参照局所性を意識する必要がある。そこで、本論文では、メモリ階層や異種メモリのパラメータの相違をアプリケーションのデータ参照局所性に適応的にマッピングするメモリ階層対応ダイナミックコンパイレーション機構を用いて実行時にプログラマから透過的に最適化する手法を提案する。本手法は、動的バイナリ変換技術を利用してランタイムの情報やプログラマから透過的にアプリケーションコードの機械語命令に近い低水準のレイヤの情報をプログラマによる手作業の解析やソースコードの修正を行うことなく取得し適応的にコード変換することにより、これまでは高度な知識を持つプログラマの手作業を必要としていたメモリ階層チューニングを自動／半自動で行うことを可能とする。本論文においては、本メモリ階層対応ダイナミックコンパイレーション機構の動作原理を述べた後、その主要な機能の一部である動的バイナリ変換技術を利用したアプリケーションプログラムのプロファイラである実行駆動型アプリケーション解析ツール *Exana* に関しての基礎的な評価結果を示し、チューニングへの応用事例についての考察を行う。

キーワード：アプリケーションプロファイリング、ダイナミックコンパイレーション、性能チューニング

A design concept of a dynamic compilation mechanism and code profiling for systems with deeper memory hierarchy

Yukinori Sato[1,2,a)]

Abstract: Locality within each of memory hierarchies must be considered in order to perform computational tasks highly efficiently on systems with heterogeneous and hierarchical memory architecture. Typically such locality is not handled well in conventional compilers and cache memory structures. In this paper, we propose a design concept of a dynamic compilation mechanism performed transparently from users, where data reference localities inherent in application program are adaptively mapped into appropriate hierarchical memory subsystems. This mechanism is implemented using a dynamic binary translation system and enables automatic/semi-automatic dynamic tuning for each memory hierarchy without reading or modifying source code. As one of primary functions of dynamic compilation mechanism, we implement an execution-driven application profiler called *Exana* and show several evaluation results.

Keywords: Application profiling, Dynamic compilation, Performance tuning

1. はじめに

近年、巨大で複雑なデータセットを解析しようという試みであるビッグデータ処理や、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。このような状況の下、アプリケーションプログラムは、更なる多機能化や複雑な事象への対応、多様な物理現象をモデルを予測するためのシミュレーションの高度化といった面で進化していくことが予想され、それらの実現のために必要と

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

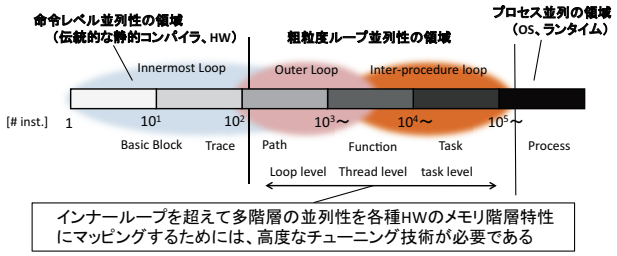

図 1 ループ階層構造

なるコード規模とその複雑性も年々増加の一途をたどると予測されている。

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が悪化する。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、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が高い処理効率を得る上で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・メニーコア化に加えて、GPU や MIC に代表される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、多階層の並列性および深いメモリ階層を意識しつつ超並列な処理を円滑に進める必要がある。

多階層の並列性および深いメモリ階層を取り扱うことにおいてループ階層構造は重要な手掛かりである。マルチグレイン・マルチレイヤにまたがる並列性を理解しループ階層を意識して性能チューニングを行うことはこれから予想されるハードウェアの深化に追従可能なソフトウェアを開発する上で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。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、アウターループ、関数を跨ぐループを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手動で抽出し、ディレクティブや API を通してプログラマが明示的にプログラムに記述する必要があり、並列化に対する知識が必要となる。同時に、対象とする HW の特徴やキャッシュメモリ特性に適切にマッピングし高い実効効率を得るためには高度なチューニングに対する技術が必要となる。すなわち、今後普及が見込まれるヘテロジーニアスで多階層なメモリアーキテクチャを持つシステムにおいて高い性能効率を得るためには既存のキャッシュメモリやコンパイラでは考慮できないメモリ階層間の特性の違いを意識した局所性の意識したチューニング技術が必要不可欠であり、更なる普及のためにはメモリ階層チューニングを自動／半自動で生産的に行う手法を開発することが広く求められている。

そこで、本論文では、メモリ階層や異種メモリのパラメータの相違をアプリケーションのデータ参照局所性に適応的にマッピングするメモリ階層対応ダイナミックコンパイレーション機構を提案し、実行時にプログラマから透過的にチューニング及び最適化を実施することを目指す。本手法は、動的バイナリ変換技術を利用して実装され、アプリケーションコードの機械語命令に近い低水準のレイヤの情報やランタイムに得られる情報をプログラマによる手作業の解析やソースコードの修正を行うことなく透過的に取得すること可能とするため、メモリ階層チューニングの生産性の問題を解決する一つの基幹技術と考えられる。

我々の研究グループではメモリ階層対応ダイナミックコンパイレーション機構の主要な機能の要素である実行駆動型アプリケーション解析ツール Exana を動的バイナリ変換技術（動的 BT) を用いて開発してきた [1] [2] [3]。本ツールはチューニング戦略を立てる上で有用となるループ階層構造に着目し、実行バイナリコード中に出現するループ階層構造単位で性能と関係する統計情報をプロファイリングすることにより処理のホットスポットの発見と、生産的な並列化を支援することを目的としている。プロファイリングにおいて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、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、プログラムの実行時に解析を行うため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、HPC の分野で分散メモリノードを用いた大規模並列処理を行う際に利用される 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。

本論文では、メモリ階層対応ダイナミックコンパイレーション機構の動作原理を述べた後、その主要な機能の一部

である動的 BT 技術を利用したアプリケーションプログラムのプロファイリングを行う機構に関しての基礎的な評価結果を示し、チューニングへの応用事例についての考察を行う。本稿では予備的な評価として、ループ変換の代表的な例であるループタイリングを実施したコードを静的解析および動的解析することにより得られた情報を示し、それらがどのようにチューニングに応用されるかを考察する。

2. メモリ階層対応ダイナミックコンパイレーション機構の動作原理

本節では、メモリ階層や異種メモリのパラメータの相違をアプリケーションのデータ参照局所性に適応的にマッピングするメモリ階層対応ダイナミックコンパイレーション機構を提案し、その動作原理を述べる。本機構は動的バイナリ変換を利用して実装され、アプリケーションコードの機械語命令に近い低水準のレイヤの情報やランタイムに得られる情報をプログラマによる手作業の解析やソースコードの修正を行うことなく透過的に取得し、それらに基づきチューニングを実施し、チューニングされた新たなコードを実行することにより実行時にプログラマから透過的にチューニング及び最適化を実施する。

2.1 メモリ階層対応ダイナミックコンパイレーション

アプリケーションプログラムの本来備えるデータ局所性だけではなく、実装上のデータレイアウトの選択やループ階層構造により様々なデータ参照特性が発生する。そこで、アプリケーションプログラムの本来備えるデータ局所性や実行するマシンの持つメモリ階層構造の特性に応じてデータ参照の局所性をランタイムに変換するコンパイラツールチェインを提案する。本機構はランタイムの情報を踏まえてループ階層構造やデータレイアウトの最適化を実施することにより自動かつ等価的にデータ参照局所性を向上させる。

図 2 に提案するメモリ階層対応ダイナミックコンパイレーション機構の概要を示す。本機構は、事前に判明しているデータサイズやシナリオに限定した最適化を実施したソースコードをコンパイラにて各種最適化オプションを指定しコンパイルした結果である実行バイナリコードを入力とし、コード実行時のランタイムの情報を踏まえたコード変換を動的に行う。実際にループ階層構造の最適化を行う手法としては、ループ展開、ループ交換、ループ融合、ループブロッキング（タイリング）といったループ変換を実施し、各ループのボディのサイズや反復数、データ転送、データ依存関係といったループのパラメタを制御する。

同時に、データレイアウトの最適化を初期配置や階層間移動に関する選択を通して実施する。データレイアウトに関する最適化はループ変換によるループ構造の最適化とも依存するため、相互にフィードバックをかける。データレ

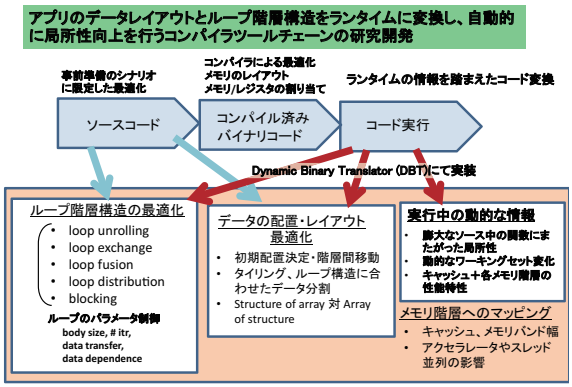

図 2 ダイナミックコンパイレーション機構の概要

イアウト最適化の結果として個々のメモリアクセスパタンがキャッシュヒットが高くなるようにコードを変換することを目指す。コード変換の例としては、Structure of Array (SoA) と Array of structure (AoS) を適宜使い分けるなどが挙げられる。

また、動的 BT で変換を行う特徴を利用し、実行中の動的な情報を活用することを目指す。具体的には、プログラムの実行時のメモリ操作の集合体として現れるワーキングセットサイズ、メモリバンド幅や連続・不規則アクセスといった動的特性をコード変換に最大限利用することやコンパイラでは解析できない大きな粒度での大域的な最適化を目指す。加えて、ハードウェアの持つメモリ階層の構造やそれぞれの階層の容量、バンド幅、レーテンシといった特性、アクセラレータと CPU のデータ移動の特性をコア内でリソースを共有する他のスレッドの影響を踏まえつつデータ局所性に最大限マッピングすることを目指す。

これれのプロセスはコンパイル時には得られないランタイムの情報を必要とするため、コンパイラが静的に完全に自動的に行うことは実現されていない。コンパイル時に得ることができない情報の代表的な例としては、ループ反復数やデータ依存関係が挙げられる。ループ反復数は入力するデータセットのサイズに由来する場合が多く、実行時に入力が決まるまでは決定できないことがしばしば発生する。また、コンパイラの行う静的依存解析はループ変換を行う上では保守的でありすぎる場合が多い。いくつかのメモリブロックの割り当てが実行時に malloc 等により行われるため、アクセスパターンに合うようにデータレイアウトを修正することは静的コンパイラにおいては困難である。

このような要因から、実際に利用されているレベルのソースコードからコンパイラがループ変換により高度に最適化されたコードを生成するのは困難であり、HPC 分野の場合、エキスパートプログラマがコンパイラに代わり手動で変換を行っているのが実情であった。加えて、マルチコア CPU においてはラストレベルキャッシュはコア間で共有されるため、実行時に同時に走っているプロセスやスレッドの影響によりリソース競合が発生する場合もある

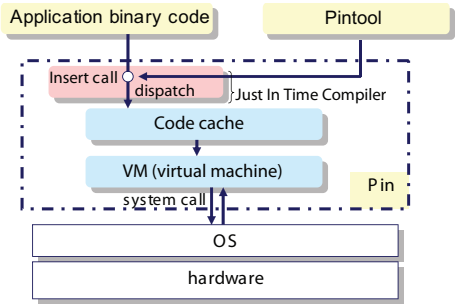

図 3 実行時プロファイリングツール Pin の概要

が、これらの影響は実行時にならないと検出することが困難であった。以上から、動的 BT でコード変換を行うことにより、既存の静的コンパイラでは達成できないレベルのチューニングが実現できることが期待される。

2.2 動作原理

本機構の構築には動的バイナリ変換（動的 BT）システムとして Pin tool set[4] を用いた。Pin はよく知られた DBT システムの 1 つであり同一命令セットへの変換を対象としている。同一命令セットへの変換においては、入力とするコードの命令セットと変換される命令セットが同一となるため、バイナリ変換をネーティブ環境で高速に実行可能である。**図 3** に Pin のソフトウェア構成要素の概要を示す。Pin は図中の点線で囲まれた VM(仮想マシン) と VM に命令を提供するためのディスパッチャとコードキャッシュから構成される。VM にて実行されるバイナリコードはコードキャッシュと呼ばれる領域に保持される。動的 BT 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インタープリタと比べて高速にコード変換を行うことが可能である。

コード変換の入力となるアプリケーションバイナリコードは Just In Time コンパイラの原理により変換されコードキャッシュに格納される。コード変換を行う区間を指定することはバイナリコードにおける任意のポイントに Instrumentation する機能により実現される。Instrumentation はコードキャッシュに対象区間のコードが格納されていない場合にのみ実行され、コードキャッシュに格納するコードを生成する。この際に、実行時に追加されるコードは解析コードと呼ばれる。解析コードはアプリケーションコードと同一のメモリ空間（プロセス）に展開され、アプリケーションコードから必要に応じて関数呼び出しを通して実行される。Pin における動的 BT においては、Pin の提供する API を通して Instrumentation のためコードおよび解析コードを事前に用意しておく必要がある。これらの 2 つのコードをは Pintool と呼ばれ、アプリケーションコードに加えて Pin を実行する際の入力となる。

Pin においては、コードキャッシュの内容操作に対する API が提供されおり、2-phase instrumentation のような柔軟なコード変換をすることが可能である [5]。2-phase instrumentation は、処理のホットスポットを特定するフェーズの後、スレッショルドを超えた区間に対して高度な最適化を実施することに用いられており、動的バイナリ変換を備えたマイクロプロセッサである Transmeta 社の Crusoe でも実装されている。Pin においては 2-phase instrumentation はフェーズを変える際に既に格納されているコードキャッシュを消去し、新しく Instrumentation し直すことにより実現される [*1]。

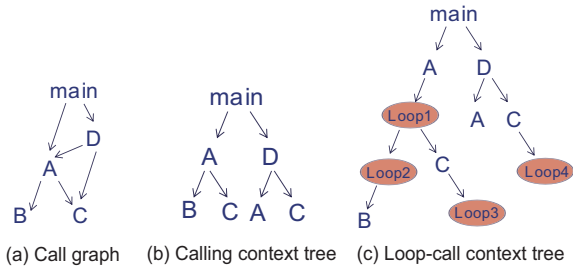

図 5 プロファイリングの手法

3. 実行駆動型アプリケーション解析ツール Exana の概要

我々の研究グループでは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てきた [1] [2] [3] [6] [7]。**図 4** に実行駆動型アプリケーション解析ツール Exana (Execution-driven application analysis 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。結果として、ループレベルの実行フロー（コンテクスト）やループ間のデータ依存関係、各ループの実行に必要なワーキングデータセットサイズを出力する。

以下、Exana の動作の流れを説明する。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解析コードを挿入する。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、解析コードが挿入された変換後の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

*1 コードキャシュへの操作が可能なのは動的 BT の特徴であり、事前にコード変換をおこなう静的 BT では実現できない

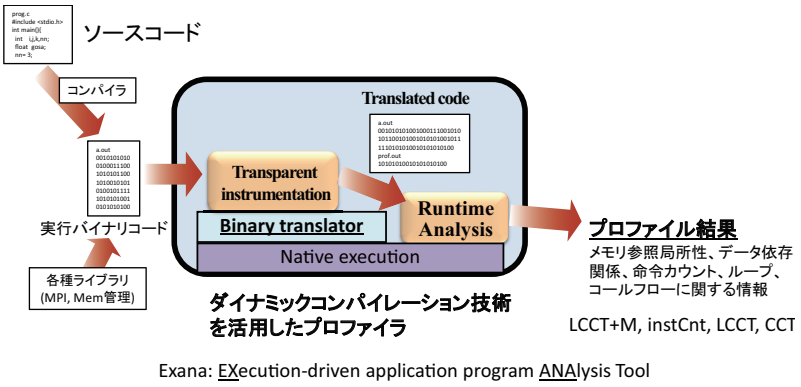

図 4　本論文で用いるプロファイラ機構

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 [8] 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 5 (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 5 (b) に CCT を示す。CCT は呼び出しシークエンスに由来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 5 (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 5 (c) の例においてはループ 1 は関数

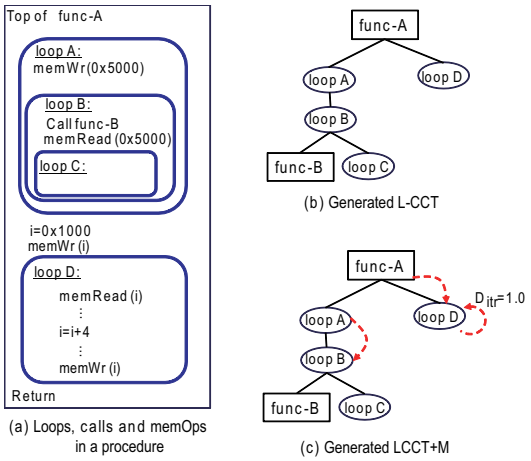

図 6　LCCT+M グラフとデータ構造

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 6** (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 6 (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 6 (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。

図 7 に密度汎関数理論にもとづく電子状態計算をオーダー N で実行するアプリケーションである OpenMX についてプロファイリングを行った際の LCCT+M の結果を示す。OpenMX 入力データには Methane.dat を用い、本結

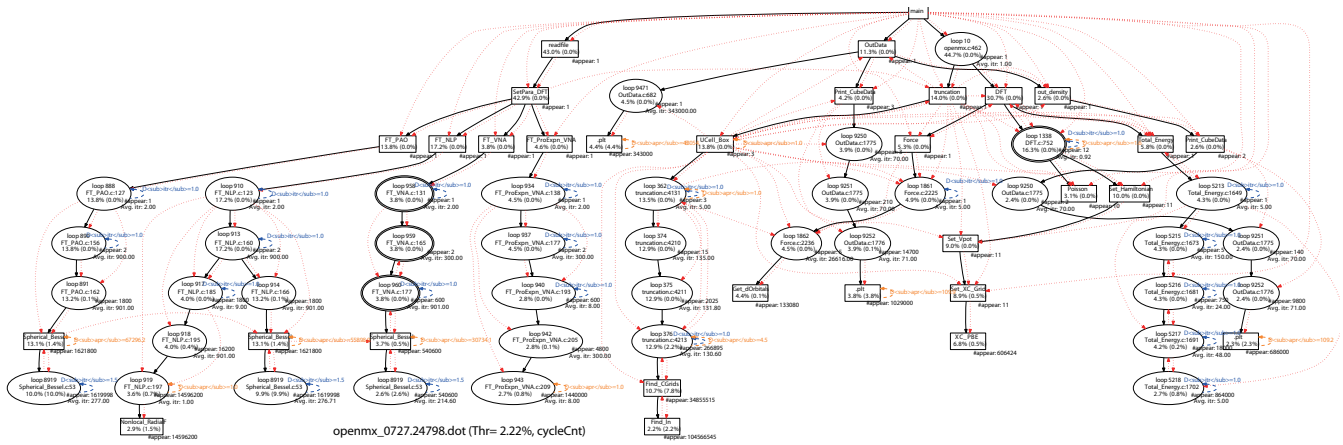

図 7　生成された OpenMX の LCCT+M

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 2.22%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。

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 行の規模であり比較的規模が大きい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入力データセットに対応するループレベルのデータの流れや実行時間が俯瞰的に理解できることが分かる。

4. プロファイリング機構の評価とチューニングへの応用

本節では、メモリ階層対応ダイナミックコンパイレーションシステムの主要な機能の 1 つであるである動的 BT 技術を利用したアプリケーションプログラムのプロファイリングを行う機構に関しての基礎的な評価結果を示し、チューニングへの応用事例についての考察を行う。評価においては HPC 分野でよく使われる行列行列積である DGEMM を題材としてループ変換の代表的な例であるループブロッキングを実施したコードを静的解析および動的解析することにより得られた結果を示し、それらがどのようにチューニングに応用されるかを考察する。

4.1 実験環境

システムの評価には以下のような環境を用いた。1 基の Xeon E5-1620 CPU と 32GB の主記憶メモリを備える構成のマシン上に OS として CentOS release 6.4 を稼働させた汎用的な x86 のサーバを用いた。この上に Pin のバージョン 2.13　(61206) 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、行列-行列積である DGEMM を C 言語にて記述したものを利用した。**図 8** にコードのカーネル部分を示す。DGEMM

```
1  #define N 4000
2  double a[N][N], b[N][N], c[N][N];
3
4  for(i = 0; i < N ; i++){
5    for(j = 0; j < N; j++){
6      for(k = 0; k < N ; k++){
7        c[i][j] = c[i][j] + a[i][k] * b[k][j];
8      }
9    }
10 }
```

図 8　行列-行列積 DGEMM のコード (Base) の概要

は以下の式にて表される。

$$C \leftarrow \alpha AB + \beta C \qquad (1)$$

本評価においては、$\alpha = \beta = 1$ とした。また、行列サイズ N は 4000 とした。こちらのベースとする実装においては i, j, k の順番でインクリメントする 3 重のループにより演算が記述されている。

4.2 ループブロッキング

ループブロッキング *2 はキャッシュメモリのデータ局所性管理メカニズムの弱点を補うために HPC 分野で広く用いられている手法である。ループブロッキングはターゲットとするマシンのメモリ階層に合わせるためにアプリケーションのデータのワーキングセットをサイズの面で調整するための変換である。特定のレベルのメモリ階層構造に合わせて再利用を最大化するようにループネスト構造の再編成とブロック（タイル）サイズの選択することによりターゲットとなるコードの局所性を最適化する。ブロッキングは、物理メモリ、仮想メモリ、キャッシュ、レジスタなど様々な異なるメモリ階層において用いることができ、データアクセスの空間および時間的局所性を強化する技術である。同様に、メモリ階層の複数のレベルを対象に同時に局

*2　ループタイリングとも呼ばれる

```
1  #define N 4000
2  #define P 32
3  #define R 32
4  #define Q 32
5  double a[N][N], b[N][N], c[N][N];
6
7   for( ii = 0; ii < N ; ii+=P){
8    for(kk = 0; kk < N ; kk+=R){
9     for(jj = 0; jj < N ; jj+=Q){
10     for(i = ii; i < MIN(N, ii+P) ; i++){
11      for(k = kk; k < MIN(N, kk+R) ; k++){
12       for(j = jj; j < MIN(N, jj+Q); j++){
13        c[i][j] = c[i][j] + a[i][k] * b[k][j];
14       }
15      }
16     }
17    }
18   }
19  }
```

図 9 ループ交換とループブロッキングを行ったコード (Opt.)

表 1 DGEMM の実行時間

	Time [s]	
	Base	Opt.
icc -O2	36.6	20.38

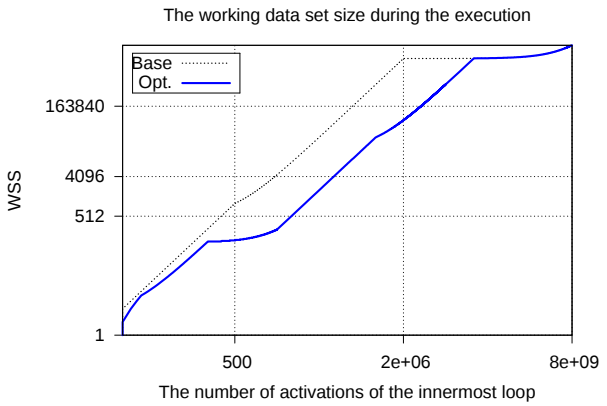

図 10 ワーキングデータセットの遷移

所性を得るために多階層にわたるブロッキングを行うことも可能である [9]。したがって、SRAM、DRAM、NVRAM のように異なる特性を持つデバイスにより構成されたメモリサブシステムには必要不可欠である。

図 9 に最適化のためにループ交換およびループブロッキングを行ったコードのアウトラインを示す。C 言語の処理系では配列を Row-major 順にメモリに格納するため、列方向の順次アクセスがシーケンシャルなメモリアクセスとなる。そこで、列方向のアクセスがより多い j を最内ループとするようにベースとする実装における j, k の順番を交換した。加えて、更なるデータ参照局所性が得られるように、32x32x32 のブロックに分割するというループブロッキングを行った。

しかしながら、これまでのコンパイラにおける静的解析のみでは、2.1 節でのべたような理由でブロッキングによる最適化がうまく達成されてこなかった。そこで、実際のプログラムの実行時にどのようにデータのワーキングセットが変化しているかをアプリケーションプロファイラにより解析を行い、実際の速度向上の結果と照らし合わせながらの検証を行う。

4.3 ワーキングデータセットの評価

Base と Opt. のそれぞれコードを Intel C++ Compiler 13.0 を用いて'-O2 -g' オプションにてコンパイルし、実行バイナリコードを生成した。これらのコードを実行した結果を**表 1** に示す。ループ交換およびループブロッキングにより実行時間が 44%短縮されている事がわかる。

データのワーキングセットが変化を観測するために、[3] にて提案したデータ依存関係を観測するためのページテーブルによりワーキングデータセットを計測する手法を用いてワーキングセットの時間的変化を調べた。本手法は、プログラムの特定のループ区間における一定の反復回転数や出現数の範囲において任意のサイズのページを基準としてワーキングデータセットを計測することが可能である。今回は、キャッシュの挙動との関連性を考察するために、観測用のページテーブルのサイズをキャッシュラインのサイズである 64B として、計算カーネル部の最内ループのボディが反復される毎にその先頭にて割り当てられたページの個数の計測を行った。

図 10 は得られたワーキングデータセットサイズの遷移を両対数グラフにて示した結果である。横軸にループボディが実行された回数、縦軸にワーキングデータセットの尺度であるページテーブルの個数 (WSS) を表す。横軸のループボディの実行された回数については、事前のプロファイリングにより図 7 に示すような LCCT+M を確認したところ、最内ループに間しては 128bit SIMD 命令が選択されていたこととループアンローリングと実施されたことによりループ反復数が 8 分の 1 となり 500 回だけ回転する事が分かった。そこで、最内ループに相当する 500 回目と、以後のループではそれぞれの次元に相当する 4000 を掛け合わせた回数に基準の補助線を設けた。縦軸のワーキングセットに関しては、キャッシュとの関連性を考慮するために以下のような対応を図った。評価に用いた CPU はコア毎に L1/L2 キャッシュを 32kB, 256kB、L3 はコア間で共有で 10MB 有する。ここで、ページあたりの容量が 64B であることからそれぞれのキャッシュ容量に対応するワーキングデータセットのサイズは 512, 4096, 163840 となる。縦軸にはこれらのサイズに対応する補助線を設けた。

結果より、ループブロッキングとループ交換が実施されるとデータの再利用が促進されワーキングデータセットの

サイズの増加スピードが緩やかになることが分かる。ワーキングデータセットサイズはこれまでに参照さえたことがないメモリアドレスのページに最初にアクセスする際にの増加される。タイリングを行わない Base の場合、j, k がそれぞれすべて実行されてた段階でおおよそ多くの配列の要素へのアクセスが完了してしまっているため一部のデータは L3 キャッシュからも落ちてしまう状況となる。一方、ブロッキングを行っている場合は、各ループネストやキャッシュ階層において局所性が高まっているため再利用による参照が増えていることが分かる。これにより、例えば、k, j のループに間しては L3 キャッシュの容量よりも小さいため、キャッシュメモリに多くのデータが格納されていることが期待できる。

以上より、ワーキングデータセットサイズの時間的な変化を監視することにより局所性の指標を定量的に得ることができることが確認された。ループタイリングにおける分割するパラメタの決定は入力となるデータサイズや実行するマシンのメモリ階層構成や実行時に割り当てられるメモリレイアウトにも依存するためコンパイルの段階では得ることが難しいと考えられている。本論文で論じたダイナミックコンパイレーション技術により更に高度なプロファイラやコード変換機構の研究開発を引き続き進め、性能チューニングをエンドユーザーから透過的かつ生産的に実行する基盤技術として開発していく計画である。

5. まとめと今後の課題

本論文では、キャッシュメモリやコンパイラでは考慮できないメモリ階層間のデータ局所性をアプリケーションのデータ参照局所性に適応的にマッピングするメモリ階層対応ダイナミックコンパイレーション機構を用いて実行時にプログラマから透過的に最適化する手法を提案した。本手法は、動的バイナリ変換技術を利用してプログラマから透過的に高度なチューニングに必要な情報を取得し、加えて、これまでは高度な知識を持つプログラマの手作業を必要としていたメモリ階層チューニングを自動／半自動で行うことを可能とする。

本論文においては、メモリ階層対応ダイナミックコンパイレーション機構の主要な機能の一部である動的バイナリ変換技術を利用したアプリケーションプログラムのプロファイリングを行う機構に関して現状の開発状況を説明し、その基礎的な評価結果を示した。更に、チューニングへの応用事例についての考察を行うため、ループ変換の代表的な例であるループブロッキングを実施したコードをプロファイリングし、それらがどのようにチューニングに応用されるかを考察した。

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。また、将来的には、データのプレースメント・レイアウトの最適化の実施や、これらの最適化を自動で行うコード最適化機構をソースコードにフィードバックするソース to ソース変換か、ランタイムなバイナリ to バイナリ変換として実装する計画である。

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] 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).

[5] Hazelwood, K. and Cohn, R.: A Cross-Architectural Interface for Code Cache Manipulation, *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 17–27 (2006).

[6] 佐藤幸紀：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，2013 年並列/分散/協調処理に関する北九州サマーワークショップ (SWoPP 北九州 2013). 情報処理学会研究報告 2013-HPC-140，pp. 1–8 (2013).

[7] 松原裕貴，佐藤幸紀：メモリトレース解析によるアクセスパターンのモデル化，2013 年並列/分散/協調処理に関する北九州サマーワークショップ (SWoPP 北九州 2013). 情報処理学会研究報告 2013-HPC-140，pp. 1–6 (2013).

[8] 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).

[9] Lam, M. S. and Wolf, M. E.: A data locality optimizing algorithm, *SIGPLAN Not.*, Vol. 39, No. 4, pp. 442–459 (2004).