Vol.2017-ARC-224 No.35
Vol.2017-SLDM-178 No.35
2017/1/25

透過的メモリ階層チューニングのための動的バイナリ変換機構の設計と開発

佐藤 幸紀[1,a)]　幸 朋矢[1]　遠藤 敏夫[1]

概要：階層化されたメモリサブシステムを持つアーキテクチャ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一般的なコンパイラでは考慮されないメモリ階層間の特性の違いを意識したメモリ参照局所性を活用する必要がある。このようなメモリ局所性チューニングをコード実行時に動的に行うことを目指して、我々の研究グループでは動的バイナリ最適化機構を研究開発している。本報告では、ループタイリングに代表されるループ変換を適応することにより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効果の予備評価を行い、実行時にプログラマから透過的にチューニングを行う機構の実現可能性を調査する。

1. はじめに

近年の計算機システムにおいて処理を消費電力および速度の面で効率化するために、汎用 CPU に加えて異種計算コアやアクセラレータ搭載によるヘテロ化、更に 3 次元積層技術を用いた多様なメモリデバイスの利用するというアプローチが急速に普及している。このようなシステム側の進化に合わせてメモリ階層は深化の一途をたどり続けている。階層化されたメモリサブシステムを持つアーキテクチャ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一般的なコンパイラでは考慮されないメモリ階層間の特性の違いを意識したメモリ参照局所性を活用する必要がある。すなわち、来るべき時代の計算機システムにおいて性能や消費電力の面での高い処理効率を達成するためには、アプリケーションに内在する多様なパターンのメモリアクセスを深化するメモリ階層向けにチューニングしていくことが鍵となると予測されている。

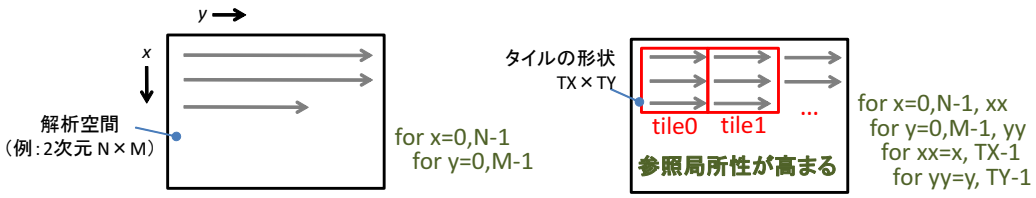
一方で、アプリケーションプログラムの構造は高機能化や高精度化に駆動される形で複雑化の一途をたどり、アプリケーションプログラムを構成するコードの量も年々増加の傾向にある。性能効率への要求が厳しい HPC 分野においては、コードチューニングは熟練の HPC 技術者がプログラムのソースコードにアクセスし、手動でプログラムの書き換えをすることにより行われてきた。しかしながら、年々多様化が進み増加の一途をたどるアプリケーションプログラムに対して、その都度ソースコードレベルでのチューニングを行うことは非常に労力がかかる作業であり、自動化や効率化が求められている。

我々の研究グループでは HPC 分野のアプリケーションコードに高度なチューニングを施すことを支援しチューニングの生産性向上を劇的に向上させることを目的として Exana ツールを開発してきた [1]。これは、動的バイナリ変換（Dynamic Binary Translation; DBT）技術を利用してコンパイルおよびリンク済みの実行バイナリコードを入力とした性能特性やメモリ参照局所性ののプロファイリングを行うツールである。Exana ツールは、コード実行時に透過的に解析コードを挿入することにより多階層構造メモリの性能特性やアクセス局所性傾向を抽出する機能を提供する。すなわち、得られた性能特性に関する統計情報を熟練した HPC プログラマが深化するメモリ階層向けのメモリチューニングに活用することに主眼を置き開発を進めてきた。しかしながら、最終的にはプログラマの手作業が必要となってしまうので、職人芸的な技術をコモディティ化するという観点からも完全な自動化が望まれる。

そこで、本報告では、メモリ局所性チューニングをコード実行時に動的に行うことを目標とする動的バイナリ最適化機構を開発することに焦点を合わせ、その実装である ExanaDBT の設計について述べた後、その基礎評価を行う。更に、ループタイリングにより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効果の予備評価を行い、実行時にプログラマから透過的にチューニングを行う機構の実現可能性を調査する。

[1] 東京工業大学　学術国際情報センター
[a)] yukinori@el.gsic.titech.ac.jp

Vol.2017-ARC-224 No.35
Vol.2017-SLDM-178 No.35
2017/1/25



(a) 2次元の場合の一般的な空間走査（2重ループ）　(b) ループタイリングを適応した空間走査（4重ループ）

図 1: ループタイリングの概要

2. メモリ局所性チューニングの現状とその自動化の試み

2.1 ループ変換によるメモリ階層チューニング

メモリ参照における空間的局所性や時間的局所性を活用するための最も一般的な手法は、ハードウェアで実装されるキャッシュメモリを用いることである。キャッシュメモリは容量の制約の下で格納されているデータの再利用を高めるために、アクセスされたデータがキャッシュに格納されていない場合、今後再利用される可能性が低いデータをリプレースし、新たにアクセスされたデータをキャッシュに格納する。しかしながら、巨大な 3 次元配列を扱う科学技術計算においては、各次元の近傍の点は時間的な参照局所性を持つことが多いが、キャッシュによる局所性管理では効果的に活用しきれていないことが知られている。このキャッシュの弱点を補うために、プログラム側のループ部分をキャッシュにヒットしやすいように変換するループタイリングを中心としたループ変換が HPC 分野ではメモリ局所性チューニングの手法として広く用いられてきた。

図 1 にループタイリング[*1]の概要を示す。ループタイリングはターゲットとするマシンのメモリ階層やアプリケーションのデータ再利用性に合わせるためにアクセスをタイル単位で進め、アプリケーションのデータのワーキングセットの時間変化量を調整するための変換である。一般的に 2 次元の解析空間を走査する場合は、図 1(a) に示されるようにプログラムにおいては 2 重ループとして記述される。y 軸方向への連続するアクセスがメモリ上の連続アクセスとなると想定すると、各次元の近傍点へのアクセスを考えた場合、y 軸方向の空間的な参照局所性は十分活用できるが、x 軸方向の参照局所性はキャッシュ容量の制約のため活用できないことが多い。そこで、図 1(b) のように、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タイルサイズと形状を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。タイリングを適応すると、2 次元の空間走査が 4 重ループにより実現される。タイル内のアクセスにおいては x 軸 y 軸の両方向の空間的および時間的な参照局所性が活用でき、参照局所性が高まる。

[*1] ループブロッキングとも呼ばれる

ループタイリングは、ストレージ、DRAM 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ことができる技術である。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り、SRAM、DRAM、NVRAM のように異なる特性を持つデバイスにより階層的に構成されたメモリサブシステムの局所性を制御する有効な手段と考えられる。

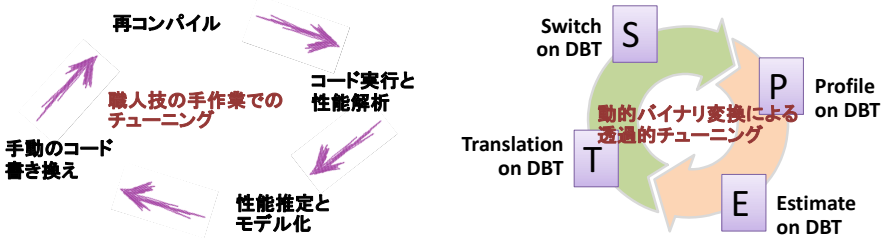
2.2 性能チューニングサイクル

HPC 分野のプログラム開発およびコードチューニングの実際の現場においては、性能チューニングはほとんどの場合、高度な技術を持つプログラマーの職人芸的な手作業のコード書き換えに依存しているのが実情である。ループタイリングを適応する場合においても同様で、多くの場合、コンパイル時にソースコードから得られる情報だけでは不確定な事象が一般的なコンパイラでの実用的な HPC コードの自動のループ変換を妨げるため、手作業でのチューニングに頼らざるを得ない状況である。

図 2(a) に典型的な HPC コードの性能チューニングサイクルを示す。チューニングは対象のソースコードをコンパイルするところからスタートする。多くの場合、この時点ではチューニングに必要な情報が十分に得られないため、コンパイラの最適化が不十分となる。そこで、実際にコードを実行してその性能解析を行い、コンパイル時にソースコードのみからでは得られない情報をプロファイリングする。プロファイリングにより得られる情報としては、ループ反復数、ランタイムのメモリレイアウトなどアプリの入力データにセンシティブな情報やコードを実行するマシンに固有なキャッシュ構成やメモリ階層構造などが挙げられる。加えて、これらが不足しているため、コンパイラのレイヤでループ変換による性能利得を精度よく見積もるモデルを作ることも困難となる。そのため、コード実行時の挙動をプロファイルすることからスタートして、性能推定とモデル化を行っていく必要がある。その後、手動のコードの書き換えを行った後、再度コンパイルを行うという一連のサイクルを性能要求をクリアするまで継続する。

マルチコア CPU においてはラストレベルキャッシュは

Vol.2017-ARC-224 No.35
Vol.2017-SLDM-178 No.35
2017/1/25



(a) 典型的なHPCコードの性能チューニングサイクル　(b) DBTに駆動される性能チューニングサイクル

図 2: 性能チューニングサイクル

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。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る半面、アプリケーション分野の経験的な知識とシステム構成に固有のチューニングに関するノウハウにより事前に解決策がなされる場合もある。いずれにせよ、NRE コスト（Non-Recurring Engineering cost）が増大し、増え続けるソフトウェア量やチューニング対象に対して持続性の面で課題となっている。また、書き換えられたプログラムはシステム構成に依存することが一般的で、その移植性にも問題がある。

図 2(b) に本研究で目指している動的バイナリ変換 (DBT) に駆動される性能チューニングサイクルを示す。本サイクルにおいては、コンパイル済みの実行形式のバイナリコードを起点に最適化がスタートする。手作業でのサイクルの各ステージに対応する処理を DBT を介して効果的に行う。各ステージにて DBT で変換を行う特徴を利用し、実行中の動的な情報を活用しつつ最適化を進めることができる。特に、コンパイル時に得ることができない情報の代表的な例であったループ反復数やデータレイアウトについての情報をプロファイリングのステージにて抽出し、後続の性能モデリングの際に利用することが可能となる。また、実行を行う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も織り込むことができる。

加えて、Exana ツールによる高度なメモリアクセス特性のプロファイリングにより、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が可能である。以上から、DBT でコード変換を行うことを核とする透過的チューニングにより、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。更に、透過的に実行コードが書き換えられるため、プログラム自体には性能を高めるための低水準の記述を行う必要がなくシステムの構成に非依存となるため、移植性を維持できる。

このように、本手法は、知的なツールチェインを組み合わせ、ハードウェアの特性を取り込んだ対象アプリケーション特化型コードを透過的に生成し、オンラインで実行を切り替えることを可能とする性能チューニングサイクルである。

2.3 関連研究

ループ最適化は、キャッシュの局所性や再利用性を向上させるためにループ反復の実行順序を造り直す変換を行う最適化である。伝統的に HPC 分野においてもタイリングを中心として様々な試みが提案されてきた。しかしながら、タイリングのサイズや形状のパラメータに関しては、ループ内の配列参照がアフィン参照に限定されることに由来しコンパイル時に決定している必要があった。これは、メモリ参照におけるすべての係数がコンパイル時に定数でないとアフィン式にはならないためである。従って、ループタイリングは、コンパイラが静的解析により試みるか、プログラマがソースコードにて明示的に最適化されたループを記述するかのどちらかにより行われることが一般的であった。一方で、DBT を用いてタイリングを含むループ変換を動的に行うというアプローチにおいては、ランタイムにタイリングのパラメータを変更することに対応できるため、応用可能な領域が広がると推測される。

DBT を用いてタイリングを含むループ変換を動的に行うというアプローチに Jain らの提案する ShapeShifter が挙げられる [2]。ShapeShifter は、Protean code という DBT システムと LLVM Polly を組み合わせ、ランタイムにタイリングパラメータを最適化したコードを生成するシステムであり、クラウド環境のようなリソースを高度に共有するダイナミックなマルチコア・マルチプロセッサの環境を対象に、同時に走るプロセスやバックグラウンドジョブの影響を踏まえキャッシュタイリングを実環境に合わせて適応していく技術である。

しかしながら、マルチプログラミング環境のような複数の独立したプログラム（タスク）を同時に走らせた際のリソースの最適な配分のためにタイリングを想定しているため、本研究で目標としている HPC コードの生産的なチューニングとは対象が異なる。HPC の分野においては、ノード内は共有されるよりも OpenMP でスレッド化されているコードですべてのコアが占有されることが一般的であり、

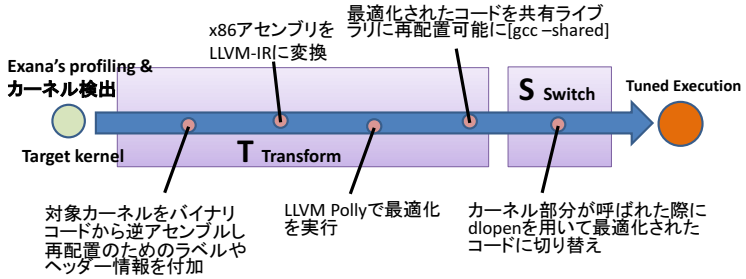


図 3: 動的バイナリ変換を用いた実行時コード最適化とコード切り替え機構

不均質なプログラムが同時に走るという ShapeShifter の想定とは大きく異なる。また、ShapeShifter はソースコードを起点に LLVM の中間言語である LLVM IR を生成することを前提としたアプローチであり、我々が目指している任意の実行バイナリを起点としてループ最適化を行っていくアプローチとは異なる。

Damschen らは DBT を用いてバイナリコード中のホットスポットをメニーコアプロセッサにオフロードする機構を提案している [3]。彼らの手法においても LLVM-Polly を用いて最適化を行われているが、想定しているバイナリコードが LLVM IR の.bc であり、x86 のコードを起点に最適化を行っていく我々のアプローチとは異なる。

3. ExanaDBT の概要

図 2(b) に示されるチューニングサイクルを実現する実行時バイナリ変換機構を用いた透過的メモリ階層チューニング機構の実装として、我々の研究グループでは ExanaDBT を開発している。ExanaDBT は、コンパイル済みのバイナリコードを入力として、最適化の対象部分をプロファイリングと性能予測モデルにより識別した後、アプリケーションのメモリアクセスパターンに適するようにループ最適化によるメモリ階層チューニングを実施したコードを生成し、実行時に最適化されたコードに切り替えることを行う。

ExanaDBT における実行バイナリ変換に基づくコード変換機構は以下のような P-E-T-S から構成される 4 つのステージからなる設計として開発を進めた [4]。P（Profile）ステージは実行時プロファイリングを行うステージであり Exana ツールとして実装してきた各種プロファイラを活用して性能チューニングに利用する統計情報を実行時に抽出する。E（Estimate）ステージは P ステージにて得られたプロファイリング情報を入力としてメモリ階層性能モデルを用いて性能利得のある最適化の対象領域や最適化の戦略を出力する。今回の実装においては、P ステージにて HotSpot 検出を行い、E ステージにおいては HotSpot として検出されたところを常に最適化するという戦略とした。

実際にバイナリコードを最適化する T（Translate）ステージについては、E ステージにて得られたチューニングによる性能利得があると判定されたカーネル部分をバイナリコード中から抜き出し、逆アセンブルした後、LLVM の IR に変換し、IR レベルで Polyhedral model に基づくループ変換を行い、最適化されたバイナリを生成するという機構の実装を行った。本実装においては既存のコンパイライインフラストラクチャの LLVM、Polly という LLVM-IR のレイヤで Polyhedral モデルに基づき最適化を行うツール [5]、や McSema という x86 のバイナリコードを LLVM-IR に変換するツール [6] といったオープンソースのツール群を活用し開発を進めた。

最適化されたコードが T ステージにて生成された後、S（Switch）ステージにおいては、コード実行をオリジナルのものから T ステージで変換したものに切り替えることを行う。本機構もバイナリ変換技術を利用して実装を行った。

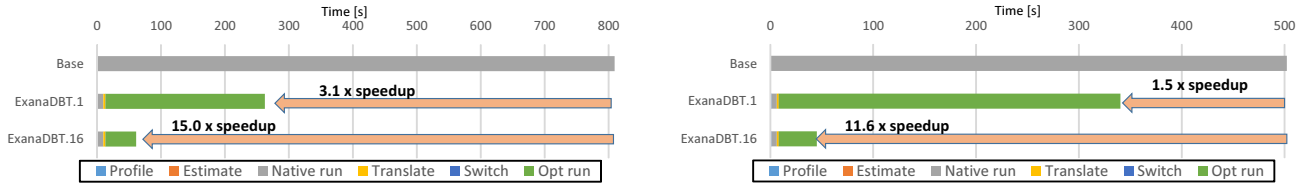
図 3 に T ステージと S ステージの詳細を示す。透過的かつ自動でチューニングが可能となる機構として実行バイナリコードからチューニング対象箇所を抽出しアセンブリ言語での表記へと変換した後に McSema により LLVM IR に変換し、Polly を用いて最適化を行う。オープンソースのツールを組み合わせることに加えて、Polly で行われる最適化に適する形式に変換できるように独自のコード変換パスを開発した。加えて、ツール間の連携を円滑とするために行った McSema についてのコードへのパッチを提供元のレポジトリに提供し、コミュニティに還元を行っている。また、Polly は LLVM-IR のレベルで Polyhedral モデルに基づく最適化によりループタイリング（ブロック化）、ベクトル化、並列化を行うことができるので、本機構により最適化を行っていない逐次の実行コードが、ループタイリング、ベクトル化、スレッド並列化が実施されたコードに透過的に変換され、これらの効果を得ることが可能である。

4. 評価

4.1 実験環境

本システムの評価には典型的な HPC システムのクラスタ構成の 1 ノードに相当する 2 基の Intel Xeon E5-2650v2 CPU と 64GB の DDR3 1866MHz DRAM からなる主記憶メモリを備え、CentOS 7.2 が OS としてインストールされている 64bit の x86 のマシンを用いた。

ベンチマークプログラムとして Polybench4.2 から 2mm



(a) 2mm　　(b) gemm

図 4: ExanaDBT を用いた透過的なチューニングによる速度向上

と gemm を用いた。これらは、Polybench の Linear algebra に分類されているプログラムであり、行列積計算カーネルである。データセットとしては LARGE を用いて、配列のデータ型として double を使用した。

Exana-DBT を実装するための DBT システムのインフラとしては Pin tool set[7] を用いた。P ステージにおける HotSpot プロファイリングのパラメータとしては 50M 命令毎にトレースのサンプリングを行い、トレースの出現が 30 を超えたものを HotSpot として検出するとした。

4.2 ExanaDBT の基礎評価

ExanaDBT を用いた透過的なコードチューニングにより性能向上を得ることが可能かどうかの実現可能性を調べるための基礎的な評価を行った。本評価実験においては、入力として、clang -O0 にてコンパイルして生成した並列化および最適化が施されていない実行バイナリを用いた。また、カーネルを検出した後、関数単位で最適化コードにスイッチすることとした。コードの入れ替えは、ホットスポットとして検出された後に改めてカーネル部分の関数が呼び出される必要があるため、今回はカーネル部分を 100 回反復するように修正した。

図 4 に ExanaDBT による透過的なチューニングによる 2mm、gemm の速度向上を示す。Base は入力としたオリジナルの逐次のバイナリコードを Native 環境で実行したものであり、ベースの逐次コードを変換し 1 スレッドおよび 16 スレッドで実行した際の時間が横軸に表されている。結果には各ステージおよび E ステージの後に再度カーネルが出現するまでに行われる Native 実行の占める時間も示されている。

第一に、本評価においては、透過的チューニングありの場合と無しの Base の出力データはビットレベルで一致していることも確認している。実行速度の面では、2mm では 1 スレッド時にベースの 3.1 倍、16 並列時に 15.0 倍となった。gemm では 1 スレッド時に 1.5 倍、16 スレッド時に 11.6 倍と 2mm よりは若干少ない速度向上となった。これらより、プロファイリングやループ変換などの DBT のオーバヘッドは十分許容できる時間に収まっていることが分かった。オーバーヘッドについては、1 スレッド時にはそれほど目立たなかったが 16 スレッド時には目立つようになっているのは、本実装においては DBT におけるコード変換前までの P ステージや E ステージは並列化されていないため一定の時間となるが、並列化による実行の高速化により、相対的に DBT のオーバーヘッドの割合が高くなったためである。

これらの結果より、ExanaDBT はコンパイル済みコードを用いた透過的なコード最適化が可能であり、性能向上に寄与することが観測された。

4.3 タイリングに関する予備評価

現状の ExanaDBT の実装においては、Polly によりループタイリング、ベクトル化、並列化についての最適化を行うことができる。これらの中で、各メモリ階層に向けて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最も大きな役割を担っているのはタイリングである。しかしながら、Polly においてデフォルトで設定されるのタイルサイズは 32x32x32 の直方体であり、各メモリ階層の特色をどの程度活用できているかが不明である。

そこで、ループタイリングにおける各次元のタイルサイズを Polly においてタイルのパラメータとして変化させ生成したコードを用いて、タイルの持つ容量や形状の違いによる性能の変化を調べた。**図 5** に gemm を 16 スレッド実行した際のタイルサイズと実行時間、メモリアクセス数、TLB ミス数、キャッシュミス数の関係を示すヒートマップを示す。ここで、タイルの形状は i,j,k の組で表し、それぞれ z,y,x 軸方向に対応する。gemm においては Polly により z 軸方向にスレッド化がされるため、i を大きくした場合にはスレッド間のロードバランスが問題になると想定される。そこで、i=8 を代表として j-k の平面に特に着目した。j の刻みは 4 からスタートした後に 64 以降は 64 間隔とした。k のサイズの間隔は 4 とした。

本計測においては、ExanaDBT は用いずにソースコードから Polly により直接生成したコードを使用した。また、CPU に内蔵の PMU のカウンタ値によりメモリアクセスの総数、L1/L2/L3 キャッシュミス数、TLB ミス数を計測した。メモリアクセス、L1 キャッシュミス数、TLB ミス数に関しては likwid-3.1.3[8] を用いて、L2/L3 キャッシュミス数に関しては IntelPCM[9] を用いて計測した。

結果より、タイルサイズを変更することにより実行時間

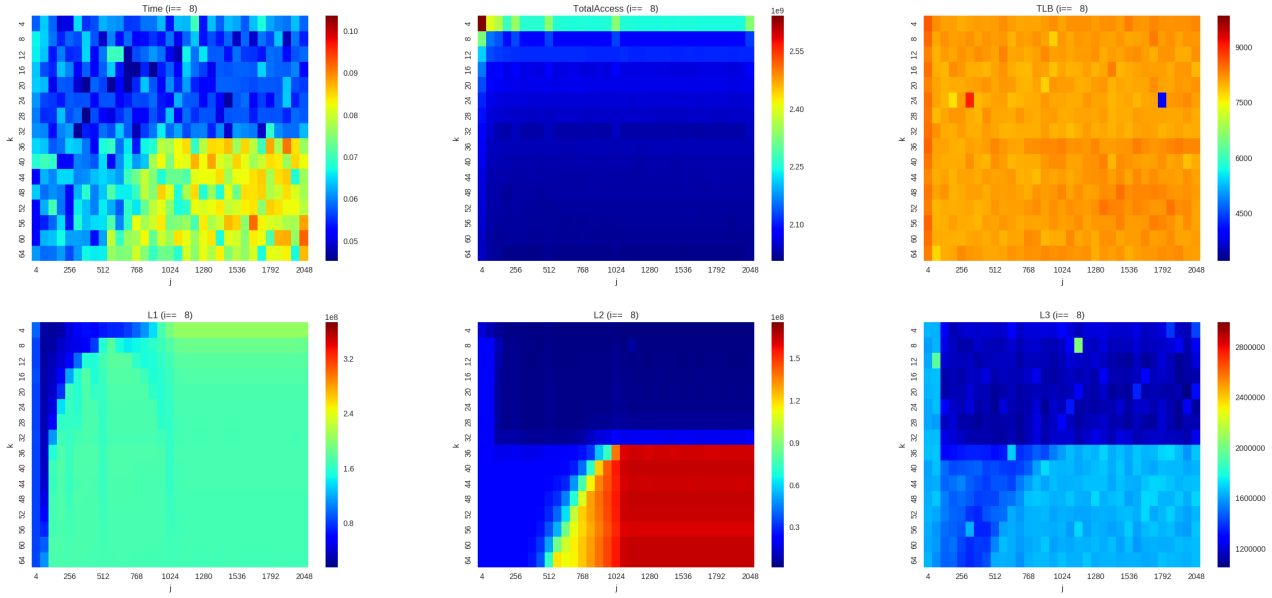


図 5: gemm のタイルサイズと実行時間、メモリアクセス数、TLB ミス数、キャッシュミス数の関係 (16 スレッド)

に最大で 2 倍程度の差があることが分かる。また、この実行時間の差には L2 キャッシュミス数が大きく関係しており、高い性能を得るためには L2 キャッシュミスを減らすことのできるタイルサイズである必要がある。さらに、L3 キャッシュミス数が少ないパラメータの組が実行時間の面でよさそうな傾向がある反面、L1 キャッシュ、TLB ミスは実行時間と相関がみられなかった。別途、Exana にて gemm のワーキングセットを計測したところ 61MB であった。従って、全体としては L3 キャッシュには収まらない容量であるが、タイリングにより L2 キャッシュを効率的に使うことができているタイルサイズは高い性能となる傾向と考えられる。一方で HW プリフェッチの影響等もあり、タイルサイズとキャッシュ構成の関係だけによる完全な性能予測は難しそうであるとの知見も得られた。メモリアクセス数に若干のばらつきがあることから、タイリングパラメータの修正に伴い生成されるバイナリコードの質が変化した可能性もあり、今後詳細に解析を行う計画である。

5. まとめと今後の課題

本報告では、メモリ局所性チューニングをコード実行時に動的に行う ExanaDBT についての設計の概要を示し、その基礎評価を行った。その結果、実行時のプログラマから透過的なチューニングにより速度向上が実現できることを示し、更に、タイリングに関するパラメータを変化させることによりメモリ階層に適した最良値が存在するという知見を得た。

今後の課題としては、タイリングサイズの最適パラメータの見積もり手法の開発と ExanaDBT において適応的に行う機構を確立していくことが挙げられる。

謝辞

本研究は、 JST 、CREST の支援を受けたものである。

参考文献

[1] Sato, Y., Sato, S. and Endo, T.: Exana: An Execution-driven Application Analysis Tool for Assisting Productive Performance Tuning, *Proceedings of the 2nd International Workshop on Software Engineering for Parallel Systems*, SEPS 2015, pp. 1–10 (2015).
[2] Jain, A., Laurenzano, M. A., Tang, L. and Mars, J.: Continuous Shape Shifting: Enabling Loop Co-optimization via Near-free Dynamic Code Rewriting, *International Symposium on Microarchitecture* (2016).
[3] Damschen, M., Riebler, H., Vaz, G. and Plessl, C.: Transparent Offloading of Computational Hotspots from Binary Code to Xeon Phi, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pp. 1078–1083 (2015).
[4] Sato, Y. and Endo, T.: Dynamic Compilation for Transparent Data Locality Analysis and Memory Subsystem Tuning, *Workshop on Architectural and MicroArchitectural Support for Binary Translation and Dynamic Optimization (AMAS-DO 2016)* (2016).
[5] Grosser, T., Groesslinger, A. and Lengauer, C.: POLLY — PERFORMING POLYHEDRAL OPTIMIZATIONS ON A LOW-LEVEL INTERMEDIATE REPRESENTATION, *Parallel Processing Letters*, Vol. 22, No. 04, p. 1250010 (2012).
[6] : McSema, https://github.com/trailofbits/mcsema.
[7] 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).
[8] : LIKWID, https://github.com/RRZE-HPC/likwid/wiki.
[9] : IntelPCM, https://software.intel.com/en-us/articles/intel-performance-counter-monitor.