Vol.2015-ARC-216 No.31
2015/8/6

実行駆動型キャッシュシミュレーションおよびメモリ参照特性解析におけるオーバーヘッドの評価

佐藤 幸紀[1,2,a)]　遠藤 敏夫[1,2]

概要：我々の研究グループではスパコンでの性能チューニングを円滑化することを目的として任意のアプリケーションコードを透過的にプロファイリングする実行駆動型アプリケーション解析ツール *Exana* を開発している。Exana ツールは、年々深化するメモリ階層向けにメモリチューニングを支援することに主眼を置き、アクセス局所性傾向を分析するとともに多階層構造メモリにマッピングした際の性能特性を推定する機能をユーザーに提供することを目標としている。現状で Exana に実装されている実行時アプリケーション解析としてはループ階層構造解析、メモリを介したデータ依存解析、メモリアクセスパターン解析、メモリ階層性能シミュレータの 4 つが主なものである。本報告では、これらの解析が実行に及ぼすオーバーヘッドをいくつかのプログラムを用いて計測することを試みる。コード実行スピードの指標である MIPS(Million Instructions Per Second) を用いて実行時間や実行命令数の異なるプログラム間においても定量的に評価が行えることを示し、実環境での実用性を考察する。

1. はじめに

近年、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のため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。一方で、HPC システムにおいては汎用 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が急速に進んでいる。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。このようなシステムにおいて高い性能を達成するためには、実際のアプリケーションコードにて発生する大量かつ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することが鍵となると予測されている。

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、コード実行時に透過的にコードの性能特性をプロファイリングする Exana ツールを開発している。Exana ツールは、年々深化するメモリ階層向けにメモリチューニングを支援することに主眼を置き、多階層構造メモリの性能特性やアクセス局所性傾向を解析する機能を提供する。

図 1 に Exana の概要を示す。Exana は動的バイナリ変換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解析を実施する。従って、利用するプラットフォーム [*1] に対応する任意のコンパイラで生成された任意の実行コードを透過的に解析できる特性を持つ。本特性により、大規模 HPC アプリケーションを解析する際も既にコンパイル済みのバイナリコードをそのまま流用する、あるいは、各種スパコン向けに記述された Makefile を修正することなく利用することができるため非常に生産性が高い。

Exana が解析の対象としているのが実行バイナリコードとそのランタイムな環境であるため、C/C++, Fortran など言語を問わず解析が可能である。従って、それぞれのコンパイラの高度な最適化の結果が反映された性能特性を解析することが可能となる。近年のコンパイラは特に SIMD に関する最適化が充実し SIMD 化に必要なループ分割やループアンローリングなどのループ変換も注力されているため、Exana を用いてコンパイラ最適化オプションとコードの品質の関係を把握することが可能となる。

Exana での解析はソースコードのないバイナリコードやライブラリの解析も可能である。一般的に HPC のコードチューニングはソースコードにアクセスすることにより行われてきた。しかしながら、年々大規模・複雑化するアプリケーションにおいて他人が書いた膨大な量のソースコードを細部まで読むことは非常に労力がかかる作業であり、

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

[*1] 現状は x86 + Linux のみに対応

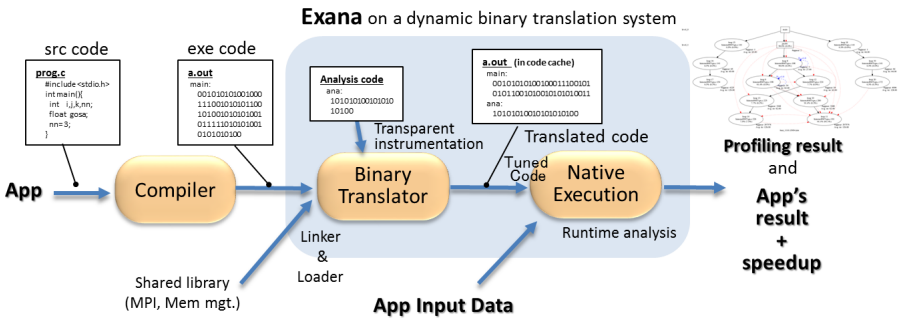

図 1　実行駆動型アプリケーション解析ツール Exana の概要

自動化が求められている。処理のカーネル部分を識別し、対応するソースコードを熟読することによりループ階層構造やデータ依存関係を理解することは可能であるが、カーネルへ入力されるデータがどのように生成されるか、また、カーネルの出力がプログラムのどの部分に利用されるかなど、プログラム全体のデータフローを把握するためには更なる時間を要する。一方で、Exana によるチューニング支援を行うことにより、ソースコードにアクセスすることなく実行のホットスポットを把握しその周辺のチューニングするべきポイントを理解する大きな手がかりとなる。この為、数百のファイルと数万行のコードから構成されるの大規模アプリケーションをチューニングする際の有用性は非常に高い。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を用いたソースコードへのフィードバックも可能である。

Exana がチューニングの支援の対象とするのは主にスパコンクラスのマシンにて実利用されているコードであるため、Exana には MPI や OpenMP を用いて記述される実用的な HPC アプリケーションコードやその実行環境において動作する実用性が備えられている。具体的には、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可能であり、OpenMP 環境、マルチスレッド環境に対応し、動的・静的リンクのどちらを用いたコードの解析にも対応する。更に、動的ライブラリがリンクされている実行コードを解析する場合においても、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。現状のバージョンでは、再帰のあるプログラム [1] の解析、バッチジョブ、子プロセスのフォークを用いたプログラムでの動作も確認している。また、Cray XC30, SGI Altix UV, TSUBAME 2.5 等のスパコン環境や汎用 x86Linux クラスタで動作が確認されるなどスパコンでの利用も実用的なレベルに達しつつあると同時に、Xeon Phi アクセラレータ向けのコードも解析できることを確認している。

Exana によるアプリケーション解析は現状では主にループ階層構造解析 [2]、メモリを介したデータ依存解析 [3]、メモリアクセスパターン解析 [4]、メモリ階層性能シミュレータ [5] という 4 つの機能から構成されている。これらの解析によりコードのチューニングに有用な情報を自動で得られる反面、解析の実施のためにはオリジナルのコード実行時間と比べて実行スピードの大幅な低下を引き起こす。スパコンクラスのマシンで動作しているコードはコード規模が大きいだけでなくコード実行時間も長時間に及ぶため、円滑な解析のためには解析ごとにおおよその実行時間のオーバーヘッドを理解しておく必要がある。

そこで、本報告では、Exana の主要な 4 つの解析が実行に及ぼすオーバーヘッドを姫野ベンチマークと NAS Parallel Benchmark のサブセットを用いて評価する。コード実行スピードである MIPS(Million Instructions Per Second) を指標として、異なるベンチマークセットやアプリケーションプログラムを通して定量的にコード実行スピードが比較できることを示し、既存のサイクル精度のシミュレータ等の MIPS 値と比較することを通して Exana の実環境での実用性を考察する。

2. チューニングを支援するコード実行特性解析

HPC 分野において性能チューニングは計算機システムのリソースを有効活用する上で非常に重要である。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特性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。今後想定されるハードウェアの進化を踏まえると、コードの中に存在する多階層のループ階層および並列性をソフトウェアの特性として理解し、それらをメモリ階層を意識してハードウェアにマッピングするという性能チューニングを行うことは鍵となる技術である。

しかしながら、ハードウェアの進化に追従可能なソフトウェアを開発するという点で生産性が極めて低い人力のチューニングに依存している状況には持続性が無く、アプリケーションに由来する多種多様な特性を自動で解析するツールの実現が急務である。そこで、我々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やそれらのメ

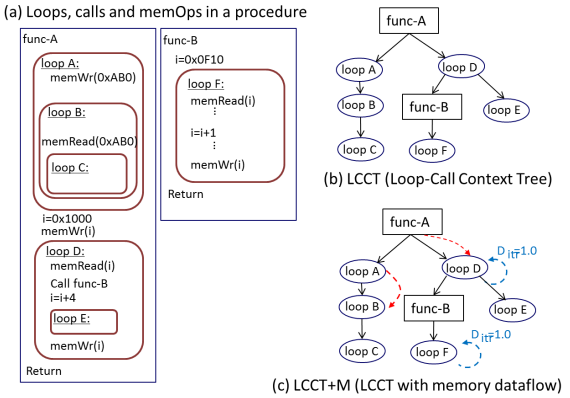

図 2　ループ階層構造解析とメモリを介したデータ依存解析の出力形式

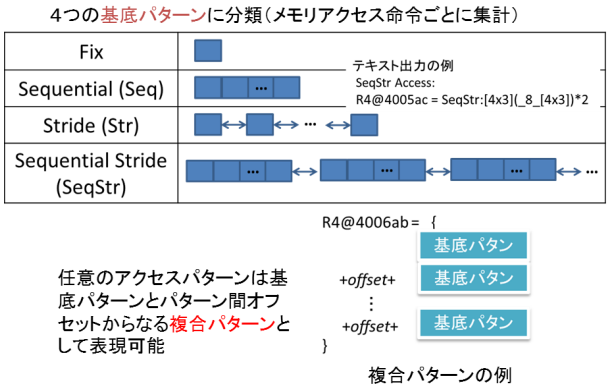

図 3　メモリアクセスパターン解析の概要

モリアクセス特性を解析する 4 つの手法を Exana の上に実装してきた。本節では、それぞれの概要を述べる。

2.1　ループ階層構造解析

ループ階層構造解析ではチューニング戦略を立てる上で鍵となるループネスト構造をプログラム実行を通して検出する [1][2]。実行バイナリコード中に出現するループ構造をバイナリコードの静的解析により検出し、コード実行時にループ区間への流入と区間からの流出を監視することにより、実際に実行されたループネストを抽出する。Exana のループ階層構造解析では、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 **LCCT (Loop-Call Context Tree)** というデータ構造を利用している。LCCT 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 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 2**(a) に関数の中にループ階層と関数呼び出しがある状況のソースの例、(b) にその場合をプロファイルした際の LCCT の出力例を示す。

LCCT はループ及び関数の呼び出し関係をまとめたコンテキストツリーであり、そのノードはループあるいは関数を示す。例えば loop A, loop B, loop C から構成される 3 重ループの場合、ループノードがアウターからインナーに向けて矢印と共にカスケードされて表示される。Exana は LCCT のノードを単位として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分析を行う。現状で実装している性能と関係する統計情報としては実行命令数、実行サイクル数、命令の種別の分布、メモリアクセス回数、メモリアクセス量などがあり、プロファイル後にインタラクティブ可視化ツール vizLcctm を用いて興味に応じて参照することができる。なお、今回の計測においては Exana の instCnt モードを利用して、命令レベルの各種統計情報の検出を常に行うとした。

2.2　メモリを介したデータ依存解析

メモリを介したデータ依存解析とは、レジスタによるデータ依存を除くメモリを介したデータ依存をループ階層構造、関数を単位として記録し、ループ並列性、タスク並列性、パイプライン並列性の推定に有用な依存情報を出力することである [3]。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 **LCCT+M (Loop-Call Context Tree with Memory)** というデータ構造を構築する。LCCT+M は LCCT にメモリ依存情報を追加したものである。図 2(c) に例示のソースコードに対応する LCCT+M を示す。

メモリを介したデータ依存関係の把握はメモリアドレス毎に最後に書き込みを行った箇所（ループなど LCCT のノードなど）を Last Write Table に保持することにより実現する。具体的には、プロファイル時にメモリアクセス命令を検出し、ランタイムに決定する参照先となるメモリアドレスを取得し、以下を実行する。

(1) メモリライト命令が実行される際、Last Write Table を更新
(2) メモリリード命令が実行される際、Last Write Table を参照し、メモリアクセス命令間の依存関係を検出。検出結果を LCCT のノードの依存リストに追加する。

加えて、ループ反復間の依存関係を検出しループ並列性を推定するためにループトリップカウントおよび出現カウントをモニタする。Last Write Table はアクセスされうる全てのメモリアドレスに対してエントリを持つ必要があるため、仮想記憶のページングの機構のようなハッシュテーブルとリストを組み合わせたデータ構造により、高速性と省メモリ性を実現する [3]。

2.3　メモリアクセスパターン解析

メモリアクセスパターン解析においては Exana でオンラインで得られるメモリアクセストレースを解析し、メモリアクセス命令毎に 4 つの基底パターン（Fix、Seqential、Stride、SequentialStride）を用いて表現する [4]。**図 3** に

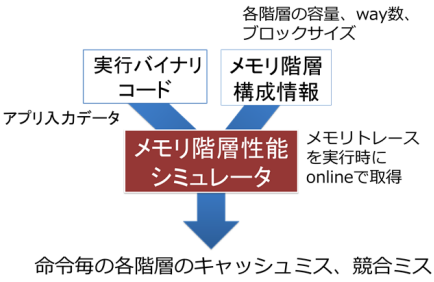

図 4　メモリ階層性能シミュレータの概要

メモリアクセスパターン解析の概要を示す。単一の基底パターンで表現ができない場合においても、任意のアクセスパターンを複数の基底パターンとそれらの間のアドレスのオフセットを用いた複合パターンとして表現することが可能である。

今回の解析においては、文献 [4] にて評価した Online 版 mempat をベースにして以下の拡張を行った。

(1) 単一の基底パターンに制限されていた Repeat の検出を 1 組の基底パターンとパターン間オフセットに拡張
(2) LCCT+M の解析と独立して mempat を解析できるように修正

メモリアクセスパターン解析においてもメモリアクセス命令毎にパターンをまとめるため、任意の数のメモリアクセス命令に対してエントリを持つ必要がある。高速性と省メモリ性を両立つつ命令毎にパターンを格納するために、LCCT+M と同様にアドレスハッシュテーブルとリストを組み合わせたデータ構造により実装を行った。

2.4　メモリ階層性能シミュレータ

メモリ階層性能シミュレータは L3 までに対応するキャッシュシミュレータを中核として HPC コードのチューニングを支援するために機能強化を行ったものである [5]。**図 4** にその概要を示す。任意のメモリ階層構成におけるメモリ性能特性を把握するために、キャッシュの階層ごとに容量や連想度を可変にできる機能を持つ。現状では、Intel Xeon の各世代の構成や京コンピュータのキャッシュ構成をシミュレーション可能なことを確認している。

HPC コードのチューニングを支援するために、「命令毎統計」という命令毎に各階層の Cache のミス率を出力する機能と、「競合解析」という命令毎に競合ミスと思われるミスを推定しその回数を出力する機能を追加している。命令毎統計は C++, SLT, map を用いて統計情報を命令毎に格納するという実装を行った。また、競合解析はキャッシュラインの競合によりデータを追い出すキャッシュ競合を検出する。キャッシュ競合は配列アクセスのときキャッシュスラッシングと呼ばれることもあり、近年の CPU キャッシュは連想度が高い（SB 世代で L1/L2=8 way, L3=20 way）が未だに発生している。競合解析はリプレースの際に LRU 落ち履歴 FIFO のエントリと比較することにより実現している。実装としては、LRU から落ちたエントリを STL vector により定義される 32 段の FIFO に保持し、リプレースされる際に FIFO に格納されているものと一致した場合を競合とみなしカウントするということを行っている。

3.　評価

3.1　実験環境

システムの評価には HP Z440 ワークステーションを用いた。本システムは 1 基の Intel Xeon E5-1630v3 CPU と 32GB の主記憶メモリから構成され、OS として CentOS release 6.6 がインストールされている x86 のサーバ環境である。ベンチマークプログラムとして姫野ベンチマークの C OMP, dynamic allocate version のサイズ S と NAS Parallel Benchmark の ft.A を使用し、それぞれスレッド数 1 にて動作させた。実行バイナリの生成には姫野ベンチマークについては Intel C Compiler 14.0.2 を、NAS Parallel benchmark については gfortran 4.4.7 を利用し、デフォルトの Makefile のオプションに-g を追加してビルドした。

Exana の 4 つの主要解析部のオーバヘッドについては、解析部のオーバーヘッドを含むプログラム実行時間と、解析対象コードの実行スピードである MIPS(Million Instructions Per Second) の双方を指標とした。オーバーヘッドの評価においては Exana(20150705 版) を用いた。メモリ階層性能シミュレータがシミュレーションを行うキャッシュ構成としては実験を行っている環境の Native の CPU に合わせるとし、L1=32kB:8way、L2=256kB:8way、L3=10MB:20way、ブロックサイズは 64B とした。また、キャッシュ構成の詳細としては、厳密な LRU を実装し、Inclusion ポリシ、ライトスルー方式で書き込みが即時にすべての階層に反映されるとした。

Exana の解析オーバーヘッドを Native 実行と比較するため CPU に内蔵の HW カウンタから likwid-3.1.3[7] の'likwid-perfctr -C 0 -g CLOCK' を用いて性能統計情報を取得した。オリジナルコードの実行命令数と実行時間である INSTR_RETIRED_ANY と Runtime unhalted の値から MIPS を算出した。

3.2　評価結果

姫野ベンチマークを用いて各機能の解析オーバーヘッドを計測した結果を**表 1** に示す。結果においては Exana の 4 つの主要解析（ループ階層構造解析、メモリを介したデータ依存解析、メモリアクセスパターン解析、メモリ階層性能シミュレータ）をそれぞれ LCCT、LCCT+M、mempat、cacheSim と表記している。

姫野ベンチマークのオリジナルコードはカーネル部分を 1 分間計測するようにカーネル部分の反復数が変動する実

表 1　姫野ベンチマークによる解析オーバーヘッドの評価

		Himeno.itr400				Himeno.1min	
		Time [s]	TimeRatio	MIPS	MIPS slowdonw	MIPS	MIPS slowdonw
native	likwid	1.48	1	4216.75	1	5493.26	1
Exana	LCCT (instCnt)	7.17	4.83	831.99	5.07	856	6.42
Exana	LCCT+M	196	132.16	30.49	138.3	38.23	62.2
Exana	mempat	91.3	61.56	65.33	64.55	62.72	87.58
Exana	cacheSim	888	598.75	6.71	628.43	9.69	566.9

表 2　MIPS 値によるオーバーヘッドの評価

		Himeno.itr400	Himeno.1min	ft.A
native	likwid	4216	5493	9713
Exana	LCCT	831.99	856	575.75
Exana	LCCT+M	30.49	38.23	98.74
Exana	mempat	65.33	62.72	136.55
Exana	cacheSim	6.71	9.69	21.05

表 3　MIPS slowdown によるオーバーヘッドの評価

		Himeno.itr400	Himeno.1min	ft.A
native	likwid	1	1	1
Exana	LCCT	5.07	6.42	16.87
Exana	LCCT+M	138.3	62.2	98.37
Exana	mempat	64.55	87.58	71.13
Exana	cacheSim	628.43	566.9	461.43

装となっている。反復数が変動すると実行される命令数が変化するためコードの実行時間の面でのオーバーヘッドを評価できない。そこで、カーネルの反復数を 400 回に固定した Himeno.itr400 を用いて時間の面でのオーバーヘッドを計測すると共に、MIPS 値がオーバーヘッドの指標となるかを検証した。

Himeno.itr400 においては Native を 1 としたときの実行時間の比である TimeRatio、及び、各解析に必要な時間を 1 としたときの Native の MIPS 値との比である MIPS slowdown（MIPS 値の逆数を取った場合の比）の双方が算出できる。MIPS 値は解析対象コードの実行命令数を基準とした実行スピードである一方で、TimeRatio は解析対象コードの実行に加えて解析処理の実行に必要なオーバーヘッドも含む値であるため必ずしも一致するものではない。しかしながら、解析対象コードの実行命令数に比例して解析処理も増加すると仮定すると、TimeRatio と MIPS slowdown は等しくなる。実際のところ、実行命令中のメモリアクセス命令の比などにも大きく変化することが想定されるため、完全一致とはならずほぼ同等の傾向を示すことが予測される。

結果より、Himeno.itr400 において TimeRatio と MIPS slowdown は、ほぼ同等の値をとることが分かった。加えて、姫野ベンチマークのオリジナルの実装である Himeno.1min において MIPS 値を計測したところ、これらとほぼ同水準の値が得られることが分かった。これらの結果より、本稿では MIPS 値および MIPS slowdown をオーバーヘッドの指標として利用することとした。

表 2 に NAS Parallel Benchmark の ft.A を含む本稿で評価を行った全てのプログラムの MIPS 値を示す。解析対象コードの特性によって、同じ解析を行った場合でも解析スピード（MIPS 値）は変化するすることが観測された。Native 実行した場合の MIPS 値においても 2 倍以上の差が発生していることを踏まえると、アプリーケーションプログラムの特性によっても解析の種類によっても CPU やメモリサブシステムにおける処理の得意不得意により MIPS 値が 2 倍程度変動することがありえるということが考察できる。

表 3 に MIPS slowdown の結果を示す。結果より、解析毎におおよそ同水準の傾向であることが観測される。また、メモリ階層性能シミュレーションのオーバーヘッドが高く Native と比べて 500 倍程度の速度低下があることが分かる。cacheSim はキャッシュ構造を模倣するシミュレーションは、LCCT+M および mempat の解析において利用されているハッシュテーブル＋リストのデータ構造より低速であることが見受けられる。

メモリ階層性能シミュレーションが特に低速である原因を検証するため、キャッシュシミュレーションの機能毎のオーバーヘッドの計測を試みる。**表 4** にキャッシュシミュレーションのオーバーヘッドの内訳を示す。cacheSim は Exana が目的とするチューニングを支援するために強化した機能である競合解析や命令毎統計に必要なオーバーヘッドも含むスコアである。競合解析 off は cacheSim から競合解析を除いたものである。Basic はチューニング支援のために強化した競合解析と命令毎統計を除いた構成で、一般的なキャッシュシミュレータに相当する。

結果より速度低下を比較すると競合解析には Basic 構成の 3 倍のオーバーヘッドが必要で命令毎統計も Basic 構成の 1.5 倍のオーバーヘッドが必要となっていることが分かる。また、Basic 構成においても Native 実行から 200 倍程度のオーバーヘッドが発生していることが観測された。

競合解析はリプレースされる際に LRU 落ち履歴 FIFO のエントリと比較を行っている。現状の実装は C++の SLT で提供される vector を用いて重複を許容する FIFO を構成しているが、C レベルの独自のデータ構造で記述するこ

表 4 Exana キャッシュシミュレーションにおけるオーバーヘッドの評価

	MIPS	slowdonw
cacheSim (チューニング支援版)	6.71	628
競合解析 off	14.19	297
Basic (命令毎統計, 競合解析 off)	19.39	217
Basic + 階層の限定（L1+L2 のみ）	22.68	186

表 5 cacheSim（Basic 版）の実行プロファイル

%	symbol name
28.2	Cache_Set::Renew_LRU()
16.8	Cache_Set::Check_Block()
16.1	RecordMem()
6.74	Cache_Set::Search_Rpl_Way()
5.84	cCache3l::process()

とにより多少の速度向上の余地はあると思われる。また、命令毎統計は C++の SLT で提供される map を用いて統計情報を命令毎に格納しているため、同様に改良の余地はあると思われる。なお、LCCT+M および mempat のハッシュテーブル及びリストは STL は使用せず C レベルでの記述にて独自に用意したものを利用している。

速度低下とメモリ階層段数の影響を調べるため、Basic 構成の階層を L1 および L2 に限定する構成での計測を行った。その結果、L3 のシミュレーションを行わない場合でも、2 割程度しか高速化しないことが分かった。これは、多くのメモリ参照が L1 や L2 のレベルで完結していると考えられる。すなわち、容量や連想度が大きく参照当たりの実行時間が大きい L3 よりも、必ず参照される L1 の処理を速めたほうが効果が高いことが示唆される。

キャッシュシミュレーションの本質的に処理が重い部分がどこになるかを識別するために Basic 構成において Linux Oprofile の operf および opreport コマンドを用いて関数単位のプロファイリングの取得を行った。**表 5** に Exana における Basic 版キャッシュシミュレーションのプロファイル結果を示す。結果より、LRU 更新が最も時間を消費しており、加えて、タグの比較や、LRU 比較も比重が大きいことが分かる。これらは全てのメモリアクセス毎に更新が必要なため、処理時間において大きな割合を占めており、大幅な速度改善は困難と推測される。

同様にチューニング支援強化版の cacheSim のプロファイルを oprofile にて取得したところ、競合解析のための FIFO 比較が最も時間を消費していることが分かった。加えて、LCCT+M および mempat のプロファイルを取得したところ、ハッシュテーブル＋リストの参照に加えて LCCT+M では依存関係を確認する関数、mempat ではアクセスパタンを決定する関数が大きな割合を占めていることがわかった。これらの結果より、LCCT+M にて実装されているデータ依存をデータ構造に格納する処理がアクセスパターンとその間のオフセットを計算するよりもオーバーヘッドは高いことが伺える。

表 6 解析スピードの比較

シミュレータ	構成	MIPS
Exana	cacheSim Avg.	12.5
CMP$im	シングルスレッドアプリ	8-12
ZSim	IPC1-NC （6 並列）	40.2
Sniper	IW-centric core model	0.45
gem5	cycle-accurate	0.25
MARSSx86	cycle-accurate	0.16

3.3 Exana と既存の CPU コアシミュレータの比較

Exana の実環境での実用性を考察するために、既存の CPU コアシミュレータの解析スピードを調査し Exana と比較することを行った。**表 6** に解析スピードを比較した結果を示す。Exana（cacheSim）の結果には、今回計測した 3 つのプログラムの平均値を用いた。CMP$im [8] は Exana と同様に Pin で実装されたキャッシュシミュレータである。Exana は CMP$im と比べて同水準の速度を達成していることがわかる。

ZSim [9] は実行をインターバルに区切り並列シミュレーションをおこなうシミュレータであり、1000 コア程度のメニーコア CPU のマイクロアーキテクチャレベルの挙動を含めた性能推定を行うことを目的としている。Exana の MIPS 値は ZSim の 6 並列での動作と比べると若干低い印象であるが、非並列の逐次実行の場合を仮定して MIPS を 6 で割ると 4.18 MIPS となり、Exana のほうが高速であることがわかる。ZSim で用いられている並列シミュレーション技術は OpenMP のプログラムを解析する際にオリジナルのスレッド数よりも大きなスレッド数を必要とするため相性が悪いと想定される。この点で、HPC アプリは OpenMP で並列化されている場合が多く、並列シミュレーションは向かないと考えられる。

他の既存のシミュレータと比較した場合、Sniper [10] のように OoO をより詳細にシミュレーションするものや、gem5 [11]、及び、MARSSx86 [12] などのような cycle-accurate のシミュレータは Exana より 2 桁程度遅いことが伺える。

以上から、Exana のメモリ階層性能シミュレータは既存のシミュレータと比べると高い水準の速度が達成されていることが分かる。また、3 つのプログラムの解析スピードの平均が mempat にて 88.2 MIPS、LCCT+M では 55.8 MIPS であることを踏まえるとメモリアクセスパターン解析とメモリ依存解析は他のシミュレータと比較して圧倒的に高速であることが分かる。

4. まとめと今後の課題

本稿では、Exana に実装されている多階層構造メモリ向

けチューニングを支援するための 4 つの主要な機能についてその解析スピードを評価した。その結果、メモリアクセスパターン解析とメモリ依存解析について平均で 88.2 MIPS, 55.8 MIPS の解析スピードであることが分かった。また、メモリ階層性能シミュレーションについては平均 12.5 MIPS であり、Native の実行と比べて平均で 552 倍程度のオーバーヘッドがあることが分かった。

代表的な Cycle-accurate simulator の解析スピードが 0.25 MIPS であることから、既存のシミュレータと比較して、HPC アプリを動作させる上で実行速度の面からも実用度が高いことが確認された。

今後の課題として、更なる解析スピード向上のための工夫が挙げられる。Native 実行が 10000 MIPS と仮定すると、100 MIPS 程度の解析で Native で 1 分間で終わるアプリの解析に 1 時間 40 分かかり、10 MIPS だと 16 時間 40 分かかってしまう。チューニング支援の観点からは必ずしも解析対象コードを全て解析しなくともチューニングに必要な情報を取得できる方法を検討していく予定である。

謝辞

メモリ階層性能シミュレータのベースとなるキャッシュの実装において多大な貢献とご協力をいただきました請園智玲博士（福岡大学）に深く感謝致します。

参考文献

[1] Sato, Y., Inoguchi, Y. and Nakamura, T.: Identifying Program Loop Nesting Structures during Execution of Machine Code, *IEICE Transaction on Information and Systems*, Vol. E97-D, No. 9, pp. 2371–2385 (2014).

[2] 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*, pp. 25:0–25:10 (2011).

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

[4] Matsubara, Y. and Sato, Y.: Online memory access pattern analysis on an application profiling tool, *International Workshop on Advances in Networking and Computing, 2014 (WANC2014)*, pp. 602–604 (2014).

[5] 佐藤幸紀，佐藤真平：メモリ階層性能シミュレータを用いた CPU 単体性能チューニング，ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集 (2015).

[6] 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).

[7] : https://github.com/rrze-likwid/likwid/.

[8] Jaleel, A., Cohn, R., Luk, C.-K. and Jacob, B.: CMP$im: A Pin-based on-the-fly multi-core cache simulator, *In Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MOBS'08)* (2008).

[9] Sanchez, D. and Kozyrakis, C.: ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems, *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp. 475–486 (2013).

[10] Carlson, T. E., Heirman, W., Eyerman, S., Hur, I. and Eeckhout, L.: An Evaluation of High-Level Mechanistic Core Models, *ACM Transactions on Architecture and Code Optimization (TACO)*, pp. 28:1–28:25 (2014).

[11] Binkert, N. et al.: The Gem5 Simulator, *SIGARCH Comput. Archit. News*, pp. 1–7 (2011).

[12] Patel, A., Afram, F., Chen, S. and Ghose, K.: MARSS: A Full System Simulator for Multicore x86 CPUs, *Proceedings of the 48th Design Automation Conference*, DAC '11, pp. 1050–1055 (2011).