Vol.2014-ARC-210 No.8
Vol.2014-OS-129 No.8
2014/5/15

細粒度時系列性能解析のための性能情報の集計手法の提案

山本昌生† 小野美由紀† 平井聡† 中島耕太†

ビッグデータ分析処理やビジネス商取引処理の高速自動化が普及している．これに伴ってプロセッサやサーバーアプリがますます高性能化，短処理化する傾向にある．このような最近の高性能システムの性能解析は従来手法のままでは極めて困難であり，システム開発者や管理者がそれらのシステムの性能最適化や性能障害原因の分析調査を容易に行うことができなくなってきている．

そこで本論文では，最近の CPU に備わっている分岐トレース支援機能を利用した性能解析手法を提案し検証する．CPU の分岐トレース支援機能を利用すれば，メモリスタック・ダンプや分岐トラップに比べてプログラムの実行トレース採取が簡単かつ低負荷になり，動作不良デバッガビリティが向上する．しかし，分岐毎の時刻情報がないために，性能デバッグにとっては情報不足である．本論文では，この問題解決のために，CPU の性能カウンタによる実測 CPI と分岐トレース機能による分岐データから，走行命令ブロック毎の実行時間算出手法を提案する．

本手法により，プログラムに手を加えずに OS からアプリまでシステムワイドに高精細な時系列性能解析が可能となる．

1. はじめに

プロセッサの高性能化が進んでいる．例えば Intel 社のサーバ向け CPU の周波数についてはサーバ用途として 1998 年に最初に登場した Pentium II Xeon が 400MHz であったのに対し，現在の Sandy Bridge 世代以降の Xeon の Turbo Boost Speed が最高 4.0GHz と 10 倍に高速化している．次にコア数について見てみると，平均しておおよそ１年半で 2 コアずつ増える傾向が見られる（図１）．最初の dual core Xeon（Paxville-DP， NetBurst 世代）が 2005 年 10 月に発表されて以降，最新の 2014 年 2 月発表の Ivy Bridge-EX（Xeon E7-2800 シリーズ）では 15cores に達しておりメニコア化が進んでいる．この様に周波数や特に最近ではコア数の面での CPU ハードの高性能化が著しい．

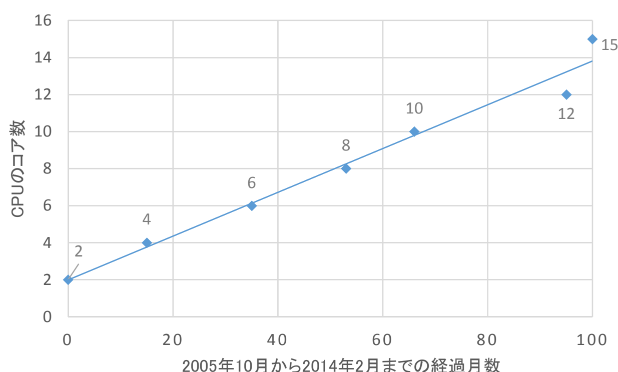

図 1: サーバ向け CPU（Intel Xeon）のコア数の増加

CPU の高性能化に対して，その上で走行するワークロードの傾向も変化してきている．従来からある OLTP（Online Transaction Processing）の様な巨大なキャッシュも駆使してシングル・プロセスの性能向上を追い求めるワークロードに対し，Internet やビッグデータ分析の普及に伴い，OLAP（Online Analytical Processing）や CEP（Complex Event Processing）などできるだけ多くのプロセスをイン・メモリで連続処理することにより性能向上を目指すワークロードが台頭してきている．さらに，CPU のメニコア化も相まって，並列分散化も進み，ワークロード環境の複雑化も進んでいる．また，Web 広告掲載をリアルタイムで高速に自動取引する RTB（リアルタイム・ビッティング）や証券取引所の売買システム上で自動高速取引を行う HFT（高頻度取引）など，極端に短いレスポンス時間を要求するワークロードも増加している．例えば RTB では，ユーザがある Web ページにアクセスしてから，広告オークションが行われ，落札した広告がそのページに掲載されるまでの応答時間が 100ms を切る．さらに，そのバックエンド・プロセスにあたる DSP（Demand Side Platform）上のプロセス処理時間は数 ms と言われている．また，プログラム同士でやり取りを行う HFT の場合は，処理時間がさらに短くなり，数十μ秒程度と言われている．この様に，ワークロードのトレンドとして，多数の連続同時実行される超短処理プロセスが台頭する様になってきている．

これらのプロセッサやワークロードのトレンドに対して，サーバ性能解析分野においては，既存手法のみに頼る解析が困難になってきている．即ち，既存手法で解析可能な環境と実環境との間に乖離が生じてきている．よって，高度化する解析対象にあわせて，性能分析技術も進化し対象環境に追いつくことが望まれている．そこで，プロセッサのメニコア化やワークロードの連続高速実行といった変化にあわせた新たなサーバ性能解析技術の創出が必要であり，我々の研究テーマとしている．特に，OS/ドライバ/ユーザプログラムに手を加えずに，低負荷で全階層の全プログラムを対象としたシステムワイドな性能解析技術が必要である．

本論文では，短処理プロセスについての新たな時間性能解析手法を提案する．対象とする短処理プロセスは処理が

† (株)富士通研究所

ⓒ2014 Information Processing Society of Japan

Vol.2014-ARC-210 No.8
Vol.2014-OS-129 No.8
2014/5/15

極めて短いため，既存のデータ採取手法では採取データが不足し，時間変化に伴う正しい挙動把握が困難となる．そこで，CPU が備える分岐トレース支援機能を利用して不足データを補完する時系列解析手法を提案する．文献[6]も分岐トレース支援機能を用いた同じ目的の提案手法であるが，分岐トレース支援機能を本研究に適用する上での課題に対する解決手法が本論文と[6]とでは異なる．本論文ではそれぞれの解決提案手法の実用化観点で特徴を比較する．

以降，２章では既存の性能解析技術や本提案手法で利用する CPU の分岐トレース支援機能とその利用課題について説明する．３章では提案手法の仕組みや実装ポイントについて述べる．４章では実機で検証した評価結果について考察する．５章では本論文の結論と文献[6]の手法との比較を述べる.そして,６章でまとめや今後の課題を記述する．

2. 性能解析手段

計算機システムの性能要因調査や計算機上で実行されるプログラムの挙動把握のために，プロファイラとトレーサの二つの性能調査方式がよく用いられる．プロファイラは，例えば数万以上の大量に採取したデータを統計処理して,プログラム上の hotspot 調査や時系列解析による挙動解析を行うことができる．特徴として，間欠的にデータ採取することにより，一般には 5%以内と採取負荷が低いことが挙げられる．一方，トレーサは対象イベントの発生毎に漏れなくデータ採取を行うために，場合によっては数十倍の負荷となるが，対象イベント処理シーケンスが完全に再現できる．以降に各性能解析手法とデータ採取方式の組み合わせによる用途の違いや課題について述べる．また，それらの用途や課題を表１として一覧にまとめる．表１中の○×の記号は，本手法の要件に合致する項目を○，要件には合わない項目を×として表している．

表 1: 性能解析手段の特徴

採取方法＼解析方法	プロファイル（統計情報解析）	トレース
サンプリング方式	・性能プロファイル（プログラムのhotspot調査や時系列による挙動把握） ・システムワイド採取が可能 ○ ・5%以下の低負荷な採取が可能○ ・採取間隔として10us以下は推奨できない（10%以上の高負荷のため）×	--
インストルメント方式	・コンパイラの最適化用入力情報採取 ・リコンパイルが必要 × ・採取対象プログラムが限られる ×	・kernel probeや関数トレーサ ・採取ポイントの事前仕込みやリコンパイル必要 × ・採取対象プログラムの制限×
トラップ方式	--	・命令トレーサ、シミュレーション用の入力情報の採取 ・全実行情報の採取が可能○ ・数十倍以上の高負荷で一般の性能調査には不向き ×

2.1 プロファイラとトレーサ

先ず，プロファイラのデータ採取方式には，サンプリング方式とインストルメント方式がある[5]．サンプリング方式は，対象イベントの一定発生毎にデータ採取トリガーを得て，その時実行されていたプログラムの実行情報や性能情報などあとの解析に必要なデータを採取する．採取トリガーの生成には OS タイマー割込みや CPU の PMC（性能モニタリング・カウンタ）のオーバーフロー割込み機能を利用するのが一般的である．性能低下の要因分析や性能最適化のためのプログラムの hotspot 調査に用いられる．課題としては,採取負荷を高くても 10%程度に抑えるためには,サンプリング間隔の短さには限界があり,現状では 10us 間隔が実質上の最短となる.一方,インストルメント方式は,プログラムの実行イメージ中に採取トリガーが静的に埋め込まれている方式で，コンパイラの最適化情報採取としてよく用いられる．短所としては，対象プログラムのリコンパイルが必要なことと，システムワイドな採取ができないことである．

次にトレーサのデータ採取方式としては，インストルメント方式とプローブ方式と CPU のトラップ方式がある.インストルメント方式の採取原理や短所は前のパラグラフのインストルメント方式と同じである．違いは，用途（解析方法）であり，主に対象とするユーザプログラムの関数トレースに用いられる．対して，プローブ方式は，採取方法自体はインストルメント方式と同じであるが，主に kernel 内に予め導入されている仕組みを指す場合が多く，プロセスのディスパッチ調査によく用いられる．長所としては，インストルメント方式と違い，採取トリガーの種類が複数用意されていて動的にどれを利用するかプログラミング可能なことと，kernel のリコンパイルは不要であることが挙げられる．しかし，OS 内の解析しかできないことが短所となる．一方，CPU のトラップ方式は，CPU のデバッグ割り込みや PMC のオーバーフロー割り込みを利用して，対象イベントが１回発生する度に割り込みをあげて，システムワイドにデータ採取を行う方式である．実行命令リタイア毎または分岐成立実行毎にデータ採取する命令トレーサとして利用される場合が多い．しかし，採取負荷が数十倍以上となり一般の性能調査には用いられることはない.主に,計算機開発時の基礎評価データ採取やキャッシュ・シミュレーション用の入力データ採取に用いられることが多い．

2.2 採取方式

以上の３つの採取方法（サンプリング方式，インストルメント方式，トラップ方式）のうち，実用的な低負荷とシステムワイドな実行プログラム情報採取が可能な点からサンプリング方式が本研究目的には適している．しかし，サンプリング方式の低負荷を担保するためには，サンプリング間隔が 10us 以上である必要がある．しかし，それでは最近の数十μ秒程度の短処理プロセスの時系列解析を行うためには情報不足となる．理想としては，対象処理に対して最低でも 100 サンプル以上は必要とみており，よって数十μ秒程度の処理に対しては数百 ns レベルのサンプリング間隔が求められることになる．しかし，それは現在の計算

Ⓒ2014 Information Processing Society of Japan

機上では困難である．そこで，従来のサンプリング方式に CPU の分岐トレース支援機能を組み合わせる方式を次章で考える．

3. 分岐情報による細粒度時系列解析手法

前章で述べたサンプリング方式の情報不足を補うために，本研究では CPU の分岐トレース支援機能を利用する方法を研究開発している[6]．即ち，最近の短処理ワークロードに対して，サンプリング方式による情報不足を CPU の分岐トレース支援機能を活用して補う方式となる．

そこで，本章では先ず，Intel CPU を例にして分岐トレース支援機能について説明し，次に本目的に Intel の分岐トレース支援機能を利用する上での障壁(時間情報が無いこと）とその解決手法を論ずる．その解決手法の提案が本論文での主目的となる．

3.1 分岐トレース支援機能と時系列解析適用への課題

最近の CPU では，成立実行された分岐命令の分岐元アドレスと分岐先アドレスの分岐ペア情報を CPU が自動的に採取してくれる機能がある．この機能を本論文では分岐トレース支援機能と呼んでいる．分岐トレース支援機能を利用すれば，メモリスタックをダンプするよりも簡単かつ確実にプログラムの実行トレース採取が可能となることが予想される．即ち，カーネル・パニック時やプログラム動作不良時のシーケンス調査によるデバッグが容易となる．しかし，分岐毎の時刻情報がないために，性能デバッグにとっては情報不足である．同じく，本論文で問題としている短処理プロセスの時系列解析に CPU の分岐トレース支援機能を用いようとしても情報不足でそのままでは利用できない．以降，本節でその問題点を説明する．

Intel 社の Intel 64/IA-32 アーキテクチャ CPU には LBR（Last Branch Record）という分岐トレース支援機能がある[1]．図 2 にその仕組みを示す．分岐元をソース，分岐先をターゲットまたはディスティネーションと言う．LBR では，16 ペア分のソースとターゲットの命令アドレス（IP: Instruction Pointer）が，専用のサイクリック・レジスタ・スタックに記録される．レジスタ・スタック上の最後の記録位置は TOS（Top Of Stack）と呼ばれる index レジスタで 0～15 の数字で示される．ソフトウェアからは，採取する分岐命令の種類（無条件分岐/条件分岐/call/return など）や特権レベル(OS モード, user モード)のフィルタリング設定，および採取開始・停止指示が可能である．フィルタリングにより，解析に無用な分岐情報は削減することができる．また，ソフトウェアからは任意のタイミングでそれらの分岐情報を読み出すことができる．

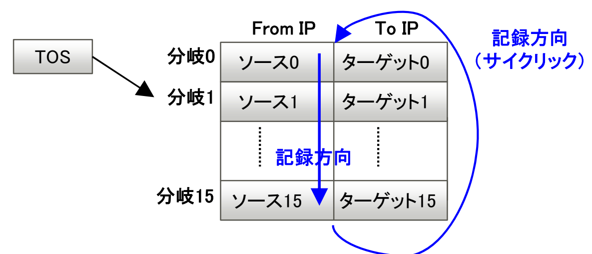

図 2: Intel LBR

さらに，Intel 64/IA-32 アーキテクチャ CPU には BTS（Branch Trace Store）と呼ばれる分岐トレース支援機能もある[1]．BTS はソフトウェアから指定した任意のメモリ領域上に CPU が自動的に分岐情報を記録してくれる機能である．LBR と比べると，メリットは記録量が多い点である．ソフトウェアからは記録先だけでなく記録領域サイズも指定できるので，採取可能な分岐データ量を増やすことができる．その反面，分岐種類のフィルタリング機能がなく，発生する分岐情報量の制御はできない．例えば LBR と違って，情報としては冗長なループ処理も採取し続けることになり大きなデメリットとなる．また，レジスタではなくメモリ上から多量のデータを読み出すことになるので，もしサンプリング毎に読み出す採取手法を採用する場合は LBR より負荷が大きくなることが容易に予想される．本研究では，実験する上で分岐種類のフィルタリングが可能な LBR を利用する．

この様に，LBR と BTS の利用にはそれぞれ一長一短があるが共通の短所もある．それは，ともに分岐毎の時刻情報がない点である．具体例を図 3 に示す．図 3 の横軸は経過時間を表す．時間軸上のブロック A,B,C,D,E は LBR などの分岐データから割り出した実行命令ブロックを示す．実行命令ブロックは，途中で分岐せずに走行し続けた命令列単位である．ここで我々の問題認識は 2 つある．(1)一つ目は，個々の命令ブロックの実行時間（または開始/終了時刻）がわからない点である．即ち，図 3 の各ブロックの長さが分からない問題である．この直接的に知ることのできない実行時間を何とか割り出そうとする試みが本論文の目的であり，解決すべき課題である．(2)二つ目は，一度に読み出す分岐データ全体（例えば，A,B,C,D,E の 5 つ）の最初の記録時刻が分からない点である．即ち，図 3 のブロック A の開始時刻(*1)が分からない問題である．これは文献[6]の手法を利用する場合に課題となる可能性がある．時刻*1 が分かっていた方で [6]の手法の精度は高くなる．なお，BTS の場合はこの問題は起きない可能性が高い．採取可能なデータ量が多いことと，オーバーフロー直前に割り込みをあげる機能もあるため，1 回の採取データ全体での開始と終了の時刻はおおよそ把握可能となり BTS のメリットの一つといえる．一方，LBR は有限のリソース上をサイクリックで上書き記録された場合は，一番古いデータの記録開始時

刻は把握できない．

そこで次節で，LBR データから抽出できる実行命令ブロックに時間情報を付与する手法を提案する．それにより LBR を使った時間ネック部の発見も迅速に可能になる．

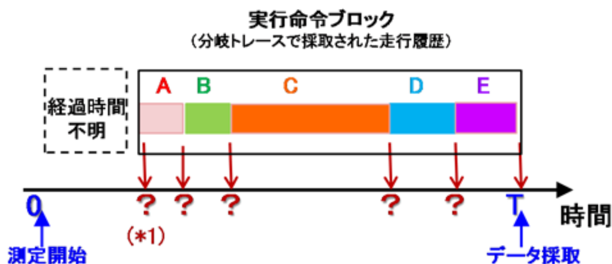

図 3: 分岐トレース支援機能の課題

3.2 実行命令ブロックの実行時間算出手法

コンピュータ・アーキテクチャの分野で知られている以下の式を利用することが基本コンセプトである[4].

CPU cycles ＝ CPI × IC （式１）
CPI：Clock cycles per instruction
IC ：Instruction count

利用する際のポイントとして，サンプリング毎に CPI を実測することにより，動的な CPI 変動も反映可能とし，より正確な命令ブロック単位の実行時間算出を可能とするところである．

図４に基本コンセプトと実装ポイントを示す

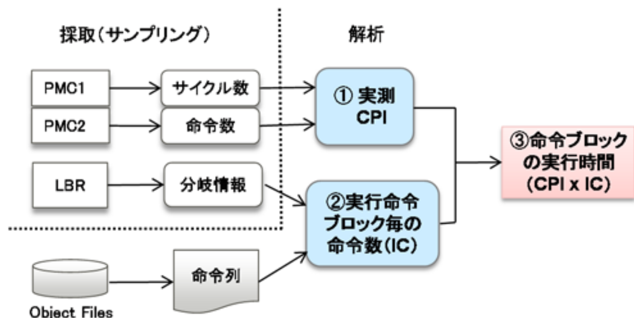

図 4: 実測 CPI と分岐情報による実行命令ブロックの実行時間算出方法

実装にあたってのポイントは３つある．

(1) CPU の PMC（Performance Monitoring Counter）を利用して，サンプリング間の CPU サイクル数と実行命令数を測定し，サンプリング毎の CPI を求める．PMC は通常 CPU に複数本用意されている．Intel 64/IA-32 CPU では，４本の汎用 PMC が用意されている．そのうちの１本をサンプリング契機用に利用し，２本を CPU サイクル数と実行命令数の測定に利用する．

(2) LBR の分岐情報と逆アセンブル・リストを照らし合わせて，サンプリング直前に実行された最大 15 個の実行命令ブロックをサンプリング毎に抽出する．図５に LBR から実行命令ブロックを抽出する方法を示す．LBR のあるスタック N-1 の分岐先アドレス N-1 から次のスタック N の分岐元アドレス N までが，命令列上を分岐せずに一直線に実行された実行命令ブロックとなる．よって，その命令区間を，開始アドレスと終点アドレスのペア情報として記録しておくことが，実行命令ブロックの抽出である．あとは，各命令ブロックの命令数を，逆アセンブル・リスト上で対応する命令区間の行数としてカウントする．

図 5: LBR 情報からの命令ブロックの抽出方法

(3) 式１より，採取された実行命令ブロック毎の実行時間を，(1)の実測 CPI と(2)の命令カウント数 IC を乗算して求める．

また，採取情報や算出値は

以上が本論文のアイデアを実装する手法となる．さらにツール化する際には，これらの採取情報や算出値を表２に示すような実行命令ブロック情報テーブルとして連続して保持管理し，解析時に利用する．

表 2: 実行命令ブロック情報テーブル例

命令ブロック	開始アドレス	終点アドレス	関数名	CPI	命令数	実行時間(Cycles)
ブロック1	ソース 1	ターゲット 1	foo1	0.81	38	30.78
ブロック2	ソース 2	ターゲット 2	foo2	1.23	7	8.61
…	…	…	…	…	…	…

4. 評価

本章では，提案手法を実機評価した結果を示す．評価の目的は二つある．①提案手法の妥当性や有効性の検証，および②Intel CPU における実用化（ツール化）の feasibility study である．

4.1 評価環境とテストプログラム

評価環境を表３に示す．また，本手法を試すためのテストプログラムとしては，SPEC CPU2006 で利用されている libquantum を利用した[2][3]．バージョンは SPEC CPU2006 にあわせて，0.9.1 を用いた．最新版は安定版が 1.0.0，開発版が 1.1.1 となっている．libquantum は，量子コンピュータをシミュレーションするプログラムでショアの多項式時間因数分解アルゴリズムを実行する．libquantum を用いた理由は，広くベンチマークとして利用されていることと多重

ⓒ2014 Information Processing Society of Japan

度をあげるとキャッシュミス率や CPI が悪くなるという特性があり本評価の要件にあっていたためである.

表 3: 実験環境（PRIMERGY RX200 S7）

CPU	Xeon™ E5-2643 3.30GHz × 2CPU/8cores/16threads
L3 Cache	25MB
Memory	90GB
OS	CentOS 6.4 (64bit)
コンパイラ	gcc version 4.4.7 20120313
最適化オプション	-O2

まず，分岐データ採取や試験解析を行う前に，対象プログラムの素性を知るためのプロファイル解析を行った．表４と表５にプロファイル結果を示す．表４は 1CPU 上で 1 スレッド実行時のプロファイル結果, 表５は 16CPU 環境で 1CPU あたり 1 スレッドをバインドして 16 スレッド実行した時のプロファイル結果を示す．それぞれ，1ms sampling rate で 30 秒間採取したデータをプロファイル解析した結果である．これらの結果を見ると，いずれも user モードで 100% 近くとなっていて，実行時間の 50% 以上を quantum_toffoli が占めていたことがわかる．また，同時実行スレッド数にかかわらず，上位関数の内訳比率がほぼ同じで変化がないことも分かる．よって，以降の検証では quantum_toffoli に着目して評価を行う．

表 4: libquantum のプロファイル（1thread on 1CPU）

Samples	%Ratio	Function	Module
17795	**59.66%**	quantum_toffoli	shor
6783	22.74%	quantum_sigma_x	shor
3631	12.17%	quantum_cnot	shor
981	3.29%	quantum_swaptheleads	shor
258	0.86%	quantum_objcode_put	shor
18	0.06%	account_user_time	vmlinux
361	1.21%	(others)	(others)
29827	100.00%	[Total]	

表 5: libquantum のプロファイル（16 threads on 16CPUs）

Samples	%Ratio	Function	Module
272775	**57.04%**	quantum_toffoli	shor
132179	27.64%	quantum_sigma_x	shor
58770	12.29%	quantum_cnot	shor
6459	1.35%	quantum_swaptheleads	shor
1472	0.31%	quantum_objcode_put	shor
495	0.10%	apic_timer_interrupt	vmlinux
6058	1.27%	(others)	(others)
478208	100.00%	[Total]	

4.2 分岐情報の採取結果

次に, 実際の LBR の採取データ例を表６に示す. 採取は, 通常のプロファイル用のサンプリング測定時に同時に LBR も採取する方法で行う. 具体的には, 1 ms sampling rate で 30 秒間採取する測定において，サンプリング毎に通常採取する情報（CPU/PID/IP/TSC など）に加え，LBR レジスタ・スタックの全情報および TOS も採取する．この測定に際して，テストプログラムはコマンドラインから「./shor 1397 8」のパラメータで予め実行させておく．また, LBR の採取対象設定として「分岐は call と return のみ採取」「特権モードは user モードのみ採取」の条件で設定した．無条件分岐を採取しなかった理由は，本パラメータ実行時の quantum_toffoli 関数処理で 100 万回以上のループ処理が発生するためである．続いて CPI を実測するために，CPU の性能モニタリングカウンタ PMC を２本用いて，CPU Cycle 数（CPU_CLK_UNHALTED.THREAD_P イベント）と実行命令数（INST_RETIRED.ANY_P イベント）も同時に採取している．これらの PMC や LBR は採取後に毎回ゼロクリアしておく．そうすると採取時に PMC には毎回，前回との差分が入っていることになる．また LBR は LBR0 から LBR15 に向けて情報が記録されていき，もしゼロのエリアがあったらまだサイクリックしていないと判断できる．

表６の採取結果を見ると，まず，命令アドレスが 0x004026b3 (quantum_toffoli)あたりを実行中にサンプリング採取が発生した時のデータであることが分かる. また, CYCLES と IC を除算して，このサンプリング区間での CPI は約 0.63 と算出できる. また, 最新の分岐情報が TOS より LBR11 にあることが分かり，かつ LBR12 以降がゼロデータだったため，サイクリックはしていないことが分かる．これは数百万回のループ実行で数 ms 程度 LBR 採取が止まっていたためである. LBR11 から LBR0 に遡ることにより call チェーンの解析ができる．LBR がアドレスのままだと分かりにくいので，マップ情報（図６）を利用してシンボル名に変換した結果を表７に示す．表７の call/return の判断は shor バイナリを逆アセンブルした命令リストから判別した．またマップ情報は shor バイナリから nm コマンドにより出力作成したものである．表７の結果から，quantum_toffoli 関数が複数の処理から頻繁に呼び出される様子が分かる. また, この call/return の関係に矛盾はなく，かつソースとの対比により正しい結果であることも確認できた．よって，期待通りにデータ採取できていることが確認できた. なお, アドレス値は 64bit であるが, 本掲載データ上の命令アドレス値の上位 32bit は全てゼロなので，省略している．

表 6: LBR も含めたサンプリングデータ例

contents of sampling data (1)		
CPU	0	
PID	5163	
IP	0x004026b3 (quantum_toffoli)	
CYCLES	3513946	
IC	5614190	
TSC	0x00147d8356ee9f8d	
TOS	11	
	from (src)	to (dst)
LBR0	0x004026cc	0x00406100
LBR1	0x00406116	0x00401b50
LBR2	0x00401b63	0x0040611b
LBR3	0x00406133	0x004026d1
LBR4	0x004026e9	0x004041ba
LBR5	0x004041b5	0x00407a90
LBR6	0x00407aba	0x004079b0
LBR7	0x004079e6	0x00402610
LBR8	0x0040263a	0x004063d0
LBR9	0x004063ea	0x0040263f
LBR10	0x0040265a	0x004070c0
LBR11	0x00407140	0x0040265f
LBR12	-	-
LBR13	-	-
LBR14	-	-
LBR15	-	-

```
・・・
shor:00000000004023f0 T quantum_sigma_x
shor:0000000000402480 T quantum_unbounded_toffoli
shor:0000000000402610 T quantum_toffoli
shor:0000000000402710 T quantum_swaptheleads_omuln_controlled
shor:0000000000402790 T quantum_cnot
shor:0000000000402850 T quantum_swaptheleads
shor:0000000000402990 T quantum_gate1
shor:0000000000403440 T quantum_r_y
shor:0000000000403560 T quantum_r_x
shor:00000000004036a0 T quantum_hadamard
shor:0000000000403750 T quantum_walsh
shor:0000000000403780 T quantum_gate2
shor:0000000000404160 T quantum_exp_mod_n
shor:0000000000404200 T quantum_qft_inv
shor:00000000004070c0 T quantum_objcode_put
shor:00000000004075d0 T quantum_objcode_stop
・・・
```

図 6: アドレスマップ例

表 7: 表６の LBR のシンボル名変換

	branch type	from	to
↓ LBR0	call	quantum_toffoli	quantum_decohere
↓ LBR1	call	quantum_decohere	quantum_gate_counter
↓ LBR2	return	quantum_gate_counter	quantum_decohere
↓ LBR3	return	quantum_decohere	quantum_toffoli
↓ LBR4	return	quantum_toffoli	quantum_exp_mod_n
↓ LBR5	call	quantum_exp_mod_n	mul_mod_n
↓ LBR6	call	mul_mod_n	muln
↓ LBR7	call	muln	quantum_toffoli
↓ LBR8	call	quantum_toffoli	quantum_qec_get_status
↓ LBR9	return	quantum_qec_get_status	quantum_toffoli
↓ LBR10	call	quantum_toffoli	quantum_objcode_put
↓ LBR11	return	quantum_objcode_put	quantum_toffoli
(current)	(sampled IP)	(quantum_toffoli)	

4.3 提案手法の検証

```
for(i=0; i<reg->size; i++){
    if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control1)) {
        if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control2)){
            reg->node[i].state ^= ((MAX_UNSIGNED) 1 << target);
        }
    }
}
```

図 7: quantum_toffoli のコアループ処理（C）

```
命令アド  機械命令          アセンブラ命令
レス(IP)  コード            リスト
------   ------------     ----------------------------------
4026a8:  48 8b 7c 08 08   mov    0x8(%rax,%rcx,1),%rdi
4026ad:  4d 89 e0         mov    %r12,%r8
4026b0:  49 21 f8         and    %rdi,%r8
4026b3:  4d 39 e0         cmp    %r12,%r8
4026b6:  75 08            jne    4026c0 <quantum_toffoli+0xb0>
4026b8:  48 31 ef         xor    %rbp,%rdi
4026bb:  48 89 7c 08 08   mov    %rdi,0x8(%rax,%rcx,1)
4026c0:  48 83 c1 10      add    $0x10,%rcx
4026c4:  48 39 f1         cmp    %rsi,%rcx
4026c7:  75 df            jne    4026a8 <quantum_toffoli+0x98>
```

図 8: quantum_toffoli のコアループ処理（アセンブラ）

提案手法による命令ブロックの実行時間算出結果の妥当性や有効性の確認を行う．妥当性の確認は，本手法による算出値と CPU の TSC（Time Stamp Counter）で実測した値を比較することにより行う．比較ベースとして quantum_toffoli 関数の実行時間を用いる．実行方法や測定条件は前節と同じである．その場合，quantum_toffoli 関数内に一つあるループ処理のループ回数は，2097152 回となることが独自に仕込んだログ採取から分かっている．図７に該当ループ処理のＣソースを示す．また，同じく該当ループ処理の逆アセンブル結果を図８に示す．ループ回数が 2097152 回の場合，本関数の実行時間や実行命令数は本ループ処理が支配的となり，該当ループ以外の実行命令数は 1%以内になるので無視する．さらに，図７のループ内の if 文が全て成立するケースを対象とすると，図８より本ループ処理は 10 命令となる．よって，以上の条件の際の quantum_toffoli 関数の実行命令数は，10×2097152 回となる．この命令数に表６の様に採取した実測 CPI を乗算することにより，ここでの条件の際の quantum_toffoli 関数の実行時間（サイクル数）を算出する．そして，そのサイクル数と TSC で測定したサイクル数を比較した結果が図９となる．1CPU に 1thread だけバインドして，CPU 数（スレッド数）を 1/8/16 の３パターンで比較を行った．その結果，いずれのパターンにおいても，TSC 値と比べて本提案手法による結果はほぼ同じとなり，本提案手法の妥当性は示せたと言える．

ⓒ2014 Information Processing Society of Japan

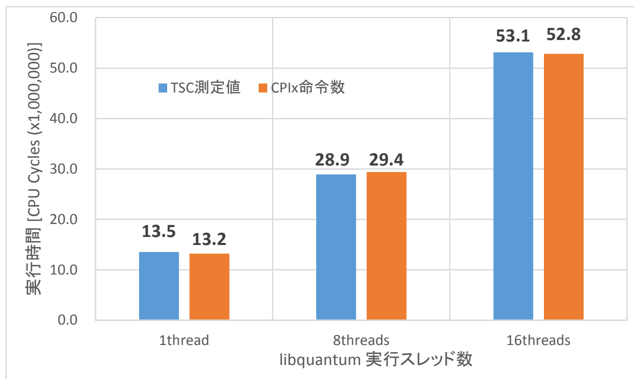

図 9: quantum_toffoli の実行命令ブロックの実行時間 [CPU Cycles (x1,000,000)]

また，スレッド数の変化に伴う実行時間の増加傾向にも正しく追従できていることも分かる．対して，式１における CPI を命令の仕様値などから算出した固定値を利用した場合はこの様な動的な変化は反映できない．仕様値はペナルティになるイベント（例：キャッシュミス）が発生しなかった場合の理想値なので，現実とは差があることが容易に予想される．しかもその差は 100 倍以上と無視できないレベルである．例えば，最近の IA システムでは，一命令あたりの実行時間が 0.3～1ns に対して，メモリレイテンシは 100ns ほどである．

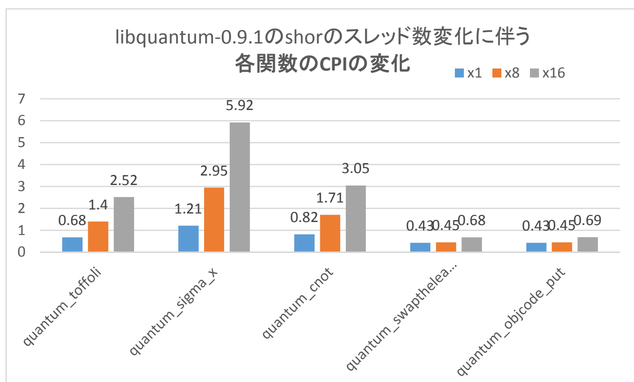

図 10: libquantum の CPI の変化

この様に，本手法は動的な変化にも対応できる有効性の高い手法といえる．この動的変化への追従は，サンプリング手法でも CPI の変化を正しく測定できることに担保されている．正しく測定できることを図 10 に示す．図 10 は libquantum のプロファイル結果の上位５つの関数のスレッド数変化に伴う CPI の変化を表したものである．この様にスレッド数変化に伴い CPI が悪化するのは，libquantum が L3 キャッシュやメモリなどのコア間で共有のリソースにアクセスし競合を起こすためである．

5. 考察

5.1 分岐トレース実用化のための CPU 機能の考察

４章の検証結果より，本手法が有効であることが確認できた．しかし，本研究の様な手法を実用化するためには，現状の CPU ではまだ機能不足である．４章の検証では内部ループ回数を別途把握した上での評価だったが，実際にはこのループ回数は利用できない要素である．

実運用サーバの性能障害要因を分岐トレースの様な機能を使って解析する場合，採取単位は関数レベルが妥当と考える．理由は，実アプリで今回の libquantum の様な数百万回もある内部ループまで採取すると，データ量的にも解析処理時間的にも無駄が多いからである．また，CPU の採取リソース量の制限もある．実際，Intel CPU の LBR の場合，採取レベルを関数単位に絞り込むことは可能だが，リソース量がかなり絞られるのでサンプリングポイントの直近しか解析できない．しかしそれでも役に立つケースはあると経験的には考えている．一方，BTS は，関数単位にフィルタリングできないので，何百万回ものループを採取してしまう. そして読み出しはメモリ上からなので LBR より高い読み出し負荷で，無駄なデータを大量に読み出すことになってしまう．それでも大量採取データの中に有効なデータも混じっている可能性はある．しかしやはり実用的とは言い難い．以上より，Intel CPU の場合，LBR と BTS で一長一短あるが, ツール化するなら LBR の方で筋が良いと感じている．理想としては，ループ処理に対するフィルタリング機能とループ回数採取機能があると有用であると考える．

5.2 実装に関する考察すべき点

表 8: 採取パターンの組み合わせ表

解析精度の向上 ⇔ 採取データ量の節約

	From & To	From	To
Call & Return	32個(16ペア)採取 [Type4]	16個採取 [Type2]	16個採取 [Type3]
Call	32個(16ペア)採取 [Type1]	--	--
Return	--	--	--

CPUリソースの節約

分岐トレース支援機能を活用する上で，表８の様な採取パターンの組み合わせも考慮すべきポイントの一つである．表８の Call と Return は LBR での採取対象の設定を表す，一方 From と To は分岐ペア情報のうち両方またはどちらかだけ採取することを表す．データ量に関する採取・解析オーバーヘッドと解析精度はトレードオフとなるので，万能型はない．よって，ケースによって臨機応変に対処することが必要と考えられる．今回の評価では，表８にある４

つのタイプを試作した．そのうち４章の評価では Type4 を用いて検証した．

5.3 文献[6]と本手法との対比

本手法は，文献[6]よりもより直接的な時間測定アプローチであるので，長所としてより正確であり動的な変化へも追従しやすいと考えられる. その反面, 制約がきつく, LBR を利用した関数単位採取の場合は，内部ループ回数が分からないので本手法では実用化できない．対して，文献[6]の手法は内部ループ回数の把握は不要なので LBR による関数単位採取にも適用しやすく, より実用性は高いと考える．

6. おわりに

本論文では，CPU の分岐トレース支援機能より得られた分岐情報と PMC で測定した実測 CPI とを活用して実行命令ブロック単位の実行時間算出手法を説明した．また，実機検証を行い，TSC による測定結果と本手法での算出値とを比較し, 妥当な結果が得られることも確認した. さらに, 仕様値などの固定値で実行時間を見積もる場合に比べて実測 CPI を用いる有効性も示した.

しかし，LBR は最大 16 ペアしか採取できずに情報量が少ないので, 今後の課題の１つとして BTS による検証も行う予定である．その際，オーバーヘッドとのトレードオフ検証が重要になる．また，より実用的にするためには，関数ブロック単位での分岐情報採取と実行時間算出が望ましい．そのためには，LBR なら call/return のみ採取する様にフィルタリングできるが，そうすると内部ループでの実行命令数がわからない. また BTS の場合はフィルタリング機能が無いため，例えば数万回の内部ループを採取する様なことが起こり，あまり実用的とは言えない．これらの課題解決手法も検討の必要がある.

今後もプロセッサの高性能化やプログラムの短処理化はまだまだ続くと思われる．よって，動的な性能チューニングや性能デバッグがますます難しくなってくることが予想される．そこで，既存のハード機能をソフト的に駆使する本論文の様なアプローチの研究開発以外に，プロセッサや OS レベルでのデータ採取の支援機能も研究開発を活発化されることを期待する．そうすれば，多くのプログラム開発者やシステム管理者の性能最適化やデバッグの面で非常に助けになると思われ，またそれによりソフトウェアが CPU の性能をより発揮できる様にもなり，好循環が創出されることが期待される.

参考文献

1) Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 3B: System Programming Guide, Part 2(253669-050US), February 2014
2) Libquantum; the C library for quantum computing and quantum simulation
http://www.libquantum.de/
3) SPEC
http://www.spec.org/
4) J. L. Hennessy, and D. A. Patterson. Computer Architecture, Fifth Edition: A Quantitative Approach, ISBN-13: 978-0123838728.
5) J. Smith, and R. Nair. Virtual Machines: Versatile Plat-forms for Systems and Processes, ISBN-13: 978-1558609105
6) 小野美由紀，他：高解像度と低オーバーヘッドを両立する時系列性能解析手法, 情報処理学会研究報告, Vol. 2014-OS-129, 2014

ⓒ2014 Information Processing Society of Japan