Vol.2014-ARC-210 No.7
Vol.2014-OS-129 No.7
2014/5/15

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

小野美由紀[†]　山本昌生[†]　平井聡[†]　中島耕太[†]

計算機の高速化により，最近では，リアルタイム性や高速性を求められる HFT（High Frequency Trading）や CEP（Complex Event Processing）などのアプリケーションサービスが広がりつつある．このようなサービスでは，ミリ秒以下で処理を完了する超高速処理が実行されている．このような超高速処理で性能劣化が発生した場合，遅延要因の特定は非常に難しい．広く用いられている性能分析ツールであるプロファイラはオーバーヘッドが少ないが，対象処理の挙動を把握するためのサンプルデータが不足する．一方，トレーサは対象処理の挙動を正確に把握できるが，オーバーヘッドが問題となる．

そこで，プロファイラと CPU が備える分岐トレース支援機構 LBR（Last Branch Record）で同時に性能データを採取し，分岐トレースデータをサンプルデータとして扱うことにより，従来のプロファイラで不足するサンプルデータを補完する手法を提案する．プロファイラで採取したサンプルデータのプロファイル結果を利用して分岐データの採取時刻を推定することにより，時刻情報を持たない分岐データをサンプルデータとして処理可能とする．本手法では，プロファイラの低負荷と分岐トレースデータによる高解像度の利点を生かすことができ，解像度を最大 17 倍に向上させた．

1. はじめに

近年の計算機の高速化により，計算機上での処理時間の短縮化が進んでいる．また，DRAM の大容量化・低価格化により，全てのデータを主記憶上に配置するインメモリコンピューティングが急速に広がりつつある．計算機の高速化により，最近では，リアルタイム性や高速性を求められる HFT（High Frequency Trading），CEP（Complex Event Processing），OLAP（online analytical processing），RTB（Real-Time Bidding）といったアプリケーションサービスが広がりつつある．

このようなアプリケーションサービスでは，ミリ秒以下で処理を完了する超高速処理が実行されている．超高速処理では，内部処理はさらに短時間で終了するため，これらの挙動を把握することは難しい．超高速処理において性能劣化が発生した場合，遅延要因の特定は非常に難しい．しかし, リアルタイム性や高速性を求められるサービスでは,数十μ秒の遅延が問題となる．広く用いられている性能分析ツールであるプロファイラはオーバーヘッドが少ないが,対象処理の挙動を把握するためのサンプルデータが不足する．一方，トレーサは対象処理の挙動を正確に把握できるが，オーバーヘッドが問題となる．

そこで, プロファイラと CPU が備える分岐トレース支援機構 LBR（Last Branch Record）で同時に性能データを採取し，分岐トレースデータをサンプルデータとして扱うことにより，従来のプロファイラで不足するサンプルデータを補完する手法を提案する．本手法により，従来手法と比較して解像度を最大 17 倍に高めることができる. 分岐トレース支援機構を利用してプロファイラの精度を上げる試みは他にはない.

本稿では, まず 2 章で既存の性能分析技術について述べ, 3 章で既存技術の課題をまとめる. 4 章でその課題を解決する提案手法, 5 章で手法の検証, 6 章では関連研究について述べ, 最後に 7 章で本手法の効果と今後の課題をまとめる.

†(株)富士通研究所

2. 性能分析技術

本章では，既存の性能分析技術について議論する．

2.1 プロファイラ

プロファイラは，動作プログラムに関するデータを大量に採取し，これを統計処理して，プログラム上で時間がかかっている部分を明らかにするものであり，性能チューニング性能劣化時の原因究明に使用される．

データの採取方法により，サンプリング方式とインストルメント方式の 2 つに分けられる．

サンプリング方式では，対象イベントが一定回数発生する毎に，動作プログラムに関するデータ採取を行う．こうすることにより，低オーバーヘッドを実現している．この方式は正確ではないが，大量のサンプルデータを解析に用いることにより, 解析結果の妥当性が統計的に担保される．図 1 のように, CPU が備える PMC（Performance Monitoring Counter）レジスタのカウンタ・オーバーフロー割込みを利用して，一定間隔毎に性能情報を採取するものが知られている. この方式では, システムワイドな採取が可能であり，対象プログラムのリコンパイルが不要という利点がある．この方式を採用しているツールとしては Oprofile などがある．Oprofiile[1]では，サンプルデータを集計し，測定時間全体におけるプロセスや関数の実行割合などの統計情報をテキスト形式で出力する．

一方，インストルメント方式では，プログラムの実行イメージに採取トリガーが埋め込まれる．そのため，採取トリガーが埋め込まれていないものは採取対象とならず，対象プログラムのリコンパイルが必要である．

ⓒ2014 Information Processing Society of Japan

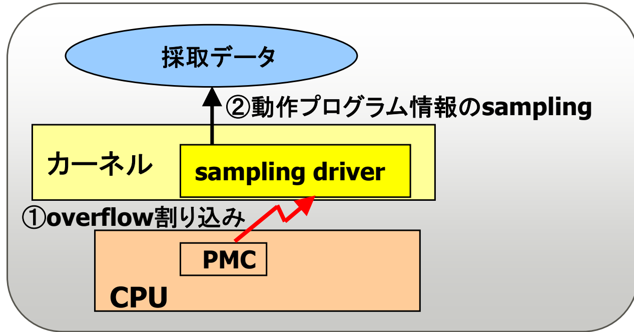


図 1　サンプリング方式ベースプロファイル手法の実装例

2.2　トレーサ

トレーサは，プログラムの実行過程を記録し，プログラムの挙動解析を行うもので，デバッグなどに使用される．対象イベントの発生毎にデータを採取するため，イベント処理を再現することができる．関数の出入り口のログを採取すれば，正確な関数の遷移が把握できる．ログとして時刻情報も採取すると，各関数の実行時間を計算することも可能である．

最近の Linux では，データ採取方法により，インストルメント方式とトラップ方式の 2 種類が提供されている．インストルメント方式は，プロファイラと同様の特徴があるが，カーネルの仕組みを使用する場合にはカーネルのリコンパイルは不要である．ただし，採取できるには OS 内部のみである．

一方，トラップ方式は，CPU のデバッグ割込みや PMC のオーバーフロー割込みを利用して，対象イベントの発生時にデータを採取する．この方式では，システムワイドにデータ採取を行えることや，対象プログラムのリコンパイルが不要であるという利点がある．

2.3　プロファイラ＋時系列解析手法

2.1 節のサンプルベースプロファイル手法で集計される統計情報だけでは，間欠的に発生する遅延要因の特定や，プログラムの挙動把握は難しい．そこで，プロファイラで採取したサンプルデータを任意の時間単位で集計し，時系列解析を行う手法が提案されている[2]．図 2 の右が 2.1 節で集計された結果を，左が時系列解析を行った結果をグラフ化したものである．左のグラフでは一定時間単位に集計した結果を時系列に表示しているため，突発的に実行割合が増えたものも見つけることができる．しかし，右のグラフでは測定時間全体を 1 つにまとめるため，一時的にしか動作しなかったものは目立たなくなってしまう．

また，時間単位の集計も行わずに，サンプルデータそのものを可視化する機能を持ったツールもある[3]．性能劣化が発生した時刻が予めわかっているような場合には，時間帯を絞り込んでサンプルデータそのものを見ることにより，問題個所を特定することができる．

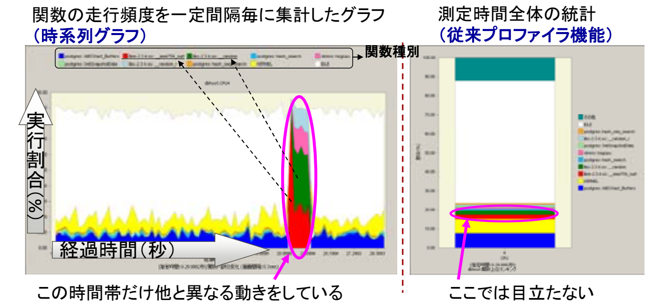


図 2　時系列解析

3.　従来技術の課題

サンプルベースプロファイラは一般的にオーバーヘッドが少ないが，サンプリング間隔を狭めればオーバーヘッドは増加していく．5 章で述べる，我々が試作したプロファイラによるオーバーヘッドの検証では，サンプリング間隔 100 μ 秒では 1%に収まるが，10 μ 秒に狭めると 11%に増加する．対象プログラムへの影響を考えると，これ以上サンプリング間隔を狭めることはできない．しかし，サンプリング間隔 10 μ 秒では 1 秒間に 100 サンプルしか採取することができない．これでは，1 ミリ秒以下で完了する超高速処理の遅延要因分析にはデータ数が十分でない．図 3 のように，遅延要因となる関数がサンプリングする間に発生していた場合，この関数はサンプリングされず，原因解明はできない．

一方，トレーサは対象処理の挙動を正確に把握できるが，オーバーヘッドが問題となる．トラップ方式では，採取負荷が数十倍以上となり，一般的な性能分析には適さない．また，トレーサではデータ量も膨大になる．

このように，サンプルベースプロファイラやトレーサでは，低オーバーヘッドに原因解明に十分なデータを採取することができないことという課題がある．

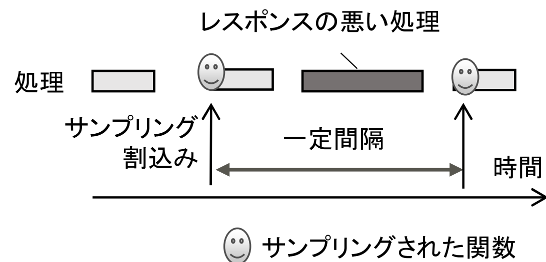


図 3　従来のサンプリング例

4.　提案手法

4.1　概要

本手法でターゲットとする超高速処理を行うアプリケ

ⓒ2014 Information Processing Society of Japan

ーションサービスでは，性能分析用データ採取のためのコードをソースに埋め込むことは望ましくない．また，効率的な分析のため，システムワイドなデータ採取が可能であることが望ましい．そこで，対象プログラムの修正を必要とせず，システムワイドなデータ採取が可能な，サンプリング方式のプロファイラと CPU が備える分岐トレース支援機構を併用する手法を提案する．提案手法では，プロファイラで性能データをサンプリングする際に分岐トレース支援機構から分岐トレースデータを採取する．採取した分岐トレースデータをサンプルデータとして扱うことにより，従来のプロファイラで不足するサンプルデータを補完する．図 4 で示すように，サンプリングの間に実行された関数を分岐データとして記録できるため，遅延要因となっている関数の特定に役立つと期待される．本手法により，プロファイラの低負荷と分岐トレースデータによる高解像度の両方の利点を生かすことができる．

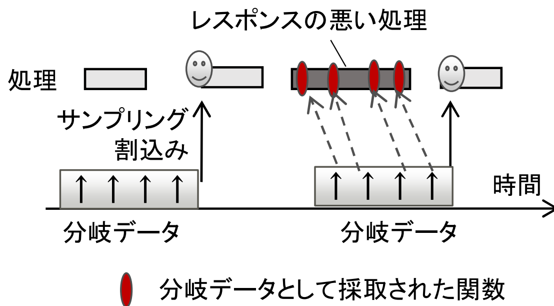
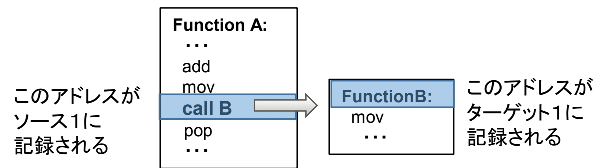


図 4　分岐トレースを使用したサンプリング例

4.2 分岐トレース支援機構

分岐トレース支援機構として Intel の CPU に備えられた LBR を使用する[4]．図 5 に LBR の原理を示す．LBR は，分岐情報として分岐元の命令アドレス（FROM_IP）と分岐先の命令アドレス（TO_IP）のペアを専用レジスタ上に複数蓄積し，あるタイミングでソフトウェアから蓄積した分岐情報を一括採取することができる．分岐情報を蓄積するレジスタの数は CPU によって異なるが，Nehalem 以降の CPU では 16 個である．レジスタスタック上の最後の記録位置は TOS（Top Of Stack）と呼ばれるレジスタに記録される．図の例では，関数 A 内の関数 B を呼び出しているアドレスが分岐元に，呼び出された関数 B の先頭アドレスが分岐先としてレジスタ分岐 1 に記録される．用意されたレジスタを使い切ると，先頭から分岐情報を上書きされ，最新情報のみが残る．レジスタに上書きする前にオーバーフロー割込みを上げる機構はないため，データ採取時に新規に記録された部分を認識できるようにしておく必要がある．ただし，分岐情報には時刻がないため，いつ分岐が発生したかはわからない．

Nehalem 以降の CPU では，採取しない特権レベルと分岐命令の種類を設定するフィルタリング機能がある．設定可能な特権レベルは OS モード/ユーザモード，分岐命令の種類は call/return/条件分岐などがある．例えば，ユーザモードの call と return 以外をフィルタリング設定することにより，ユーザモードの call と return のみを採取することができる．



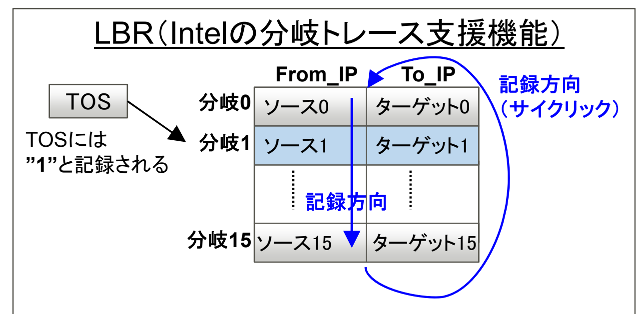


図 5　LBR の原理

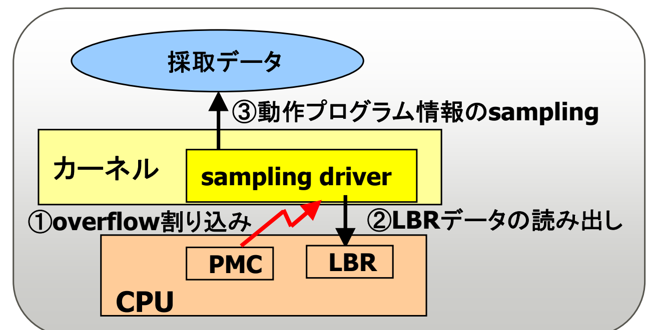


図 6　提案手法におけるデータ採取の仕組み

図 6 に本手法におけるデータ採取の仕組みを示す．指定されたサンプリング間隔毎に PMC からオーバーフロー割込みが発生する．この割込みをサンプリングドライバで受け取り，性能情報を採取する．このとき，同時に LBR レジスタから前回の割込み後に蓄積された分岐情報を読み出す．サンプルデータと分岐データに時刻情報を付けてメモリ上に記録し，サンプリング終了後にファイルに出力する．

4.3 採取データ

LBR で採取するデータについて考える．レジスタの数は限られており，有効なデータを採取するためには採取種類の絞り込みが必要である．OS 関数については Ftrace などのカーネルの機能を利用して関数の遷移を調べることができる．そこで，既存ツールでは低負荷に関数の遷移を調べることが難しいユーザ関数のみを採取対象とする．また，

ⓒ2014 Information Processing Society of Japan

遅延要因を特定するためには，なるべく正確に実行された関数の遷移が採取できたほうがよい．図 7 の関数遷移パターンはネストした関数呼出しと 1 つの関数からの複数関数呼出しを組み合わせたものである．採取したログのみからこのパターンの正しい関数遷移を得るためには，call と return を採取することが必要である．call と return を採取する場合でも，採取データを分岐元あるいは分岐先のどちらか一方に絞ることができれば，採取するデータ量を削減できる．この場合には，最後の関数あるいは最初の関数を採取することができなくなる．通常，分岐先の関数は，次の分岐命令の分岐元関数になる．従って，最新の分岐先関数はサンプリングされた関数と同じになるはずなので，分岐元データのみを採取することにする．

このように分岐データとして採取する分岐種類をフィルタリングすることによって，サンプルデータと同時に採取した分岐データから，サンプリングされた関数が呼び出された関数遷移の正確な把握を可能にする．

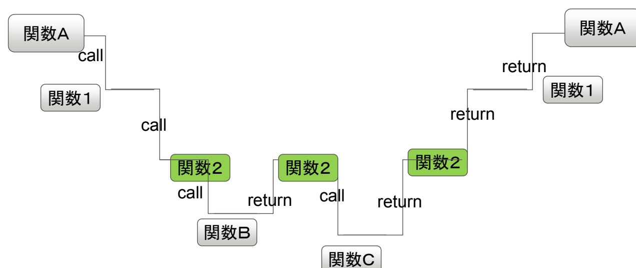


図 7　関数遷移パターン

4.4 分岐データの時刻

さらに，時刻情報を持たない分岐データの発生時刻（分岐した時刻）を推測できれば，サンプルデータとして扱うことができ，従来のサンプリング間隔で採取したデータの補完が可能になる． この分岐データへの時刻割当方法についても検討しており，時刻割当方法として 3 種類を考えている．なお、サンプルデータも含めて考える．

1）サンプリング間隔を採取した分岐データに均等に割り当てる．

2) サンプルデータのプロファイル結果から分岐データに出現した各関数の実行割合を利用し，サンプリング間隔を関数の実行割合に応じて各分岐データに配分する．配分されない分はサンプルデータに加算する．

3) 2)と同様にサンプルデータのプロファイル結果から分岐データに出現した各関数の実行割合を利用し，同時に採取された分岐データ内の関数の実行割合比に応じてサンプリング間隔を配分する．

図 8 にこの 3 種類の割当方法と分岐データを採取していないサンプルデータのみの場合 (0) の違いを簡単な例で説明する．図の例では，3 つの関数 F1, F2, F3 が順番に呼び出され，各実行割合を 10%:30%:10%とする．

0) サンプルデータのみでは，サンプリングされた 1 つの関数しか見えない．

1) LBR データを単純に均等割にすると，実際の割合と合わない．

2) LBR データを実行割合で配分すると，配分されない分がサンプル関数に加算され，サンプル関数の重みが大きくなる．LBR データの割合は妥当である．

3) LBR データを実行割合比で配分すると，実際の動きに近くなる．

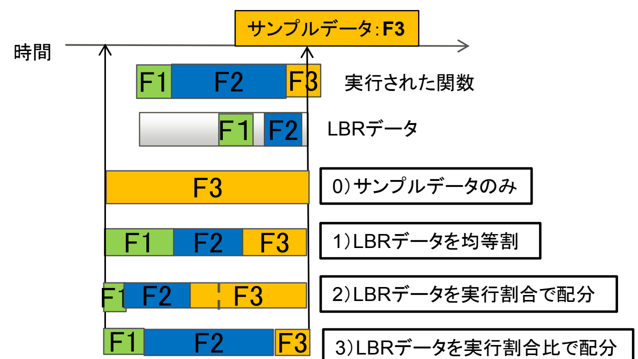


図 8　時刻割当方法による見え方の違い

5. 検証

本手法を試作し，オーバーヘッド，解像度，サンプルデータの補完の 3 点を検証した．

5.1 オーバーヘッド

プロファイラ単体と，LBR を使用したプロファイラのオーバーヘッドを検証した．検証は，プログラムの実行時間を測定し，プロファイラ非起動時の実行時間と比較することにより行った．検証環境を表 1 に示す．実行したプログラムは，関数遷移の確認のために作成した，短時間処理を行う複数の関数をランダムに実行する単純なものである．

表 1　検証環境

CPU	Sandy Bridge (XeonE5/Corei7-3930K)
OS	Red Hat Enterprise Linux Server release 6.3
Kernel	Linux version 2.6.32 -279.el6.x86_64

プロファイラと LBR 使用版プロファイラはそれぞれサンプリングレート 100μ秒と 10μ秒の 2 パターンを測定し，合計 4 パターンについて，プロファイラを起動しない通常実行と比較する．time コマンドでプログラムの実行時間を 5 回ずつ計測し，5 回の平均値を比較する．（標準偏差を平均値で割った変動係数は 0.002~0.03 であり，ばらつきは大きくない．）

通常実行時の実行時間で正規化した結果を図 9 に示す．

Ⓒ2014 Information Processing Society of Japan

“100us”はプロファイラでの100μ秒間隔でのサンプリング,“10us”は10μ秒間隔でのサンプリング,“LBR”はLBR使用版を表している．プロファイラの 100μ秒間隔でのサンプリングは通常実行に比べて 1%のオーバーヘッドに収まっている．10μ秒間隔になると，11%に増加する．LBR使用版プロファイラでは LBR レジスタの読み込みなどの処理の追加により，LBR を使用しないプロファイラよりもオーバーヘッドが増加する．しかし，LBR 使用版の 100μ秒サンプリングのオーバーヘッド 5%は，LBR 未使用プロファイラの 10μ秒サンプリングよりもオーバーヘッドが低い．LBR 使用版では，サンプルデータに対して最大 16 個の分岐データを採取できることを考えると，より少ないオーバーヘッドで解像度を高くできると言える．

今回の試作では，関数遷移を正確に把握するために，call と return を採取するように設定した．そのためにオーバーヘッドが高くなっていると思われる．採取対象を call あるいは return のみに絞り込み，採取データから関数遷移を推測することも考えられる．これにより，オーバーヘッドを削減できると思われる．また，関数遷移の確認しやすさのため，プログラム内の各関数は，次の関数を呼び出すだけのものや簡単な計算のみを行うもので，非常に短時間で終了するようにした．このため，LBR レジスタへの書込みはかなり発生したと考える．より実際アプリに近いもので検証することが望ましい．

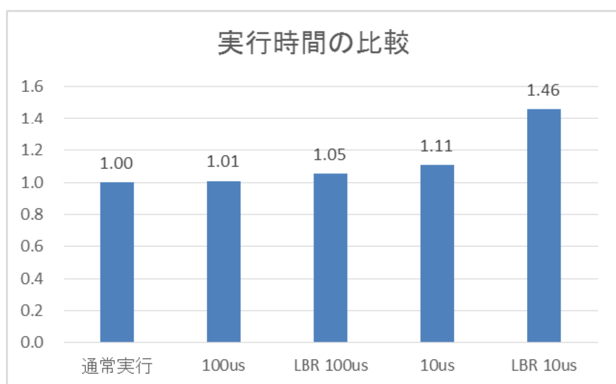


図 9　実行時間によるオーバーヘッド比較

5.2 解像度

LBR 使用版プロファイラでは，サンプルデータに対して最大 16 個の分岐データを採取できる．つまり，17 倍の点が取れることになる．超高速処理では関数レベルの実行時間が短く，１つのサンプルデータに対して採取される分岐データの数は最大 16 個になる場合が多いと予想される．従って，最大の効果が得られると期待される．

解像度の検証のため，簡単な関数遷移テストプログラムを作成し，実際に関数の遷移が分岐データとして採取されていることを確認した．図 10 にテストプログラム（左）と採取データ（右）のイメージを示す．テストプログラムは前章で取り上げた関数遷移パターンを C 言語で記述したものである．採取データは関数 p3_C がサンプリングされた際の分岐データの一覧であり、16 個採取できている．FROM0 が最新，FROM15 が最古の分岐元関数であり，図の下から上へと時間は経過している．関数 p3_C は FROM0 の関数 P3_f2 から呼ばれており，FROM5 から辿ると，関数 p3 から実行された関数の遷移がわかる．FROM15 から FROM6 までで関数 p3 から順番に関数を実行し，p3 に戻る遷移がわかる．このように，LBR データから関数実行の遷移を確認することができる．

```
p3() {
  for(i=0; i<100; i++)
{
      p3_A();
  }
}
p3_A() {
   p3_f1();
}
p3_f1() {
   p3_f2();
}
p3_f2() {
   p3_B();
   p3_C();
}
p3_B() {
    …
}
p3_C() {
     …
}
```

```
CPU 2 PID 21408 p3_C
  FROM0   p3_f2
  FROM1   p3_B
  FROM2   p3_f2
  FROM3   p3_f1
  FROM4   p3_A
  FROM5   p3
  FROM6   p3_A
  FROM7   p3_f1
  FROM8   p3_f2
  FROM9   p3_C
  FROM10  p3_f2
  FROM11  p3_B
  FROM12  p3_f2
  FROM13  p3_f1
  FROM14  p3_A
  FROM15  p3
```

図 10　テストプログラム（左）と採取データ（右）

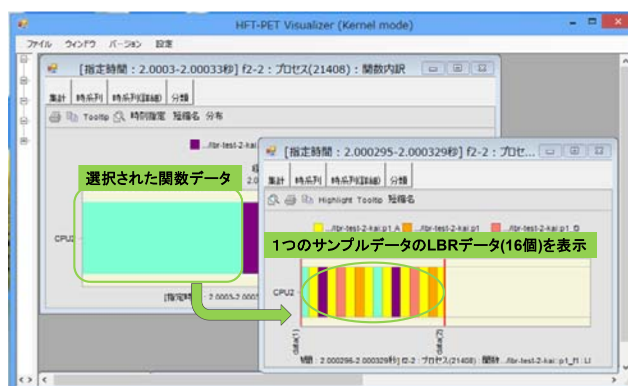


図 11　サンプルデータに対する分岐データ表示例

試作したツールには採取データを可視化するツールも含まれており，サンプルデータと分岐データを見る機能も試作した．図 11 がその表示例であり、横軸は経過時間を表す．まず，気になる関数データを選択する．選択したサンプルデータに分岐データが存在すれば，別ウィンドウに分岐データが左側から採取順に表示される．このように，GUI で関数の遷移を確認することができる．

ⓒ2014 Information Processing Society of Japan

アプリケーション実行時に採取したログなどから遅延が発生した時刻がわかっている場合には，遅延発生時刻あたりのデータを可視化ツールで確認することにより，容易に遅延原因関数を特定することができる．

5.3 サンプルデータの補完

分岐データに時刻割当を行い，分岐データをサンプルデータとして扱えるようにすると，可視化ツールで見ることが可能になる．図 12 は同じ採取データに対して，異なる時刻割当方法を使用してサンプルデータを補完したものの見え方を比較した例である．表 2 は図 12 に出現する関数の実行割合である．図 12 では関数が色分けされており、表の列の色はこれに対応している。

0) 分岐データを使用していない通常データでは，サンプリングされた関数しか見えない．1) サンプリング間隔を分岐データ数で均等割りした場合にはすべてが同じ時間間隔で採取されたように見える．2) 関数の実行割合で配分すると，配分されない分がサンプル関数に加算され，サンプル関数の重みが大きく見える．3) 関数の実行割合比で配分すると，関数の実行割合が反映され，実行割合の高い関数がよりはっきり見えるようになる．4.4 節で述べたように見えており，実際の動作に近いと推測されるが，妥当性の検証が必要である．

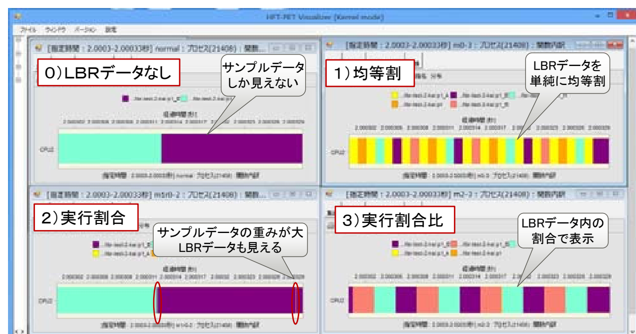


図 12 時刻割当方法による見え方の違い

表 2 関数の実行割合

サンプル数	実行割合	関数名
26283	0.33	p1_f3
25957	0.32	p1_f1
25609	0.32	p1_f2
49	0.00	p1_A
17	0.00	p1

6. 関連研究

分岐トレース支援機構を使用した 2 つの試みがある．LBR ではない別の分岐トレース支援機構である BTS（Branch Trace Store）を使用した分岐トレーサがある[5]．分岐トレーサは，Linux の perf ツールのサブコマンド branch として実装される．BTS は任意サイズのメモリ領域に分岐トレースデータを記録するものである[4]. LBR と比較すると，BTS はより多くのデータを記録できるが，フィルタリング機能（ユーザ/OS のみ）が不十分であり，レジスタではなくメモリ領域を使用するため，オーバーヘッドが大きくなる．本手法でも BTS の利用により，漏れのない分岐データ採取が可能になるが，オーバーヘッドと分岐データへの時刻割当が課題となる．

もう 1 つの試みとして，本手法と同様に PMC に加え，LBR を使用する研究がある[6]．この研究では，仮想環境においてハイパーバイザーから仮想マシン内で動作するアプリの命令レベルのモニタリングを行うことを目的としている．PMC ベースのモニタリングにおいて，割込み時に発生する採取漏れを補うために LBR を使用する．採取する仕組みは似ているが，利用目的が異なる．

7. おわりに

本稿では，プロファイラと CPU が備える分岐トレース支援機構 LBR で同時に性能データを採取し，分岐トレースデータをサンプルデータとして扱うことにより，従来のプロファイラで不足するサンプルデータを補完する手法を提案した．本手法により，サンプルデータに対して最大 17 倍の解像度を実現できることを示した．

今後の課題として，オーバーヘッドの削減がある．関数遷移を正確に把握するために call と return を採取するようにしたことがオーバーヘッドの要因の 1 つと考えられる．採取対象を call あるいは return のみに絞り込み，分岐元データに加えて分岐先データも採取することにより、採取されていない関数遷移を採取データから特定することが考えられる．また，分岐データへの時刻割当方法については，さらに効果や使い分け方の検証と改良が必要である．

参考文献

1) Oprofile: http://oprofile.sourceforge.net/about/
2) 山村周史, 他: 時系列データの統計解析による PC クラスタシステム解析手法の提案，情報処理学会論文誌. コンピューティングシステム 47(SIG_12(ACS_15)), pp.250-261 (2006)
3) 小野美由紀，他: 性能データの可視化分析ツール，情処理学会第 72 回全国大会 1-43,44 (2010)
4) LBR: CHAPTER 16 DEBUGGING, PROFILING BRANCHES AND TIME-STAMP COUNTER, Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A:System Programming Guide, Part 1, June 2010
5) Nagai, A.: Introduce New Branch Tracer 'perf branch', LIUNUXCON JAPAN 2011, http://events.linuxfoundation.org/slide/2011/linuxcom-japan/lcj2011_nagai.pdf
6) Sebastian, V. and Claudia, E.: Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture, EuroSec'12

ⓒ2014 Information Processing Society of Japan