平成 14 年度未踏ソフトウェア創造事業、OLTP 性能向上のためのメモリプロファイリングツール

A Memory Profiling Tool for Performance Tuning of OLTP Workloads

吉岡 弘隆
Hirotaka Yoshioka

ミラクル・リナックス株式会社
（〒107-0052 東京都港区赤坂４－１－３０　E-mail: hyoshiok @ miraclelinux.com ）

ABSTRACT. The performance of CPU has been improved more than 50 % per year but the performance of memory access latency is much lower than those of CPU. Therefore the penalty of memory access miss has been increased every year. Recent studies show optimization on scientific workloads is not effective on commercial workloads like DBMSs. We have developed a memory profiling tool to collect performance information of application program and OS kernel and evaluated the effectiveness of the tool.

表 1: Intel Xeon 2GHz/Main Memory 1GB での実測値

メモリ階層	アクセスコスト
L1	1 nsec
L2	9 nsec
Main memory	196 nsec

1 背景

ＣＰＵの処理速度は年率数十%で向上しているが、メモリの処理速度の向上は年率数%といわれており、ＣＰＵの向上率に比較して低い。(図 1)

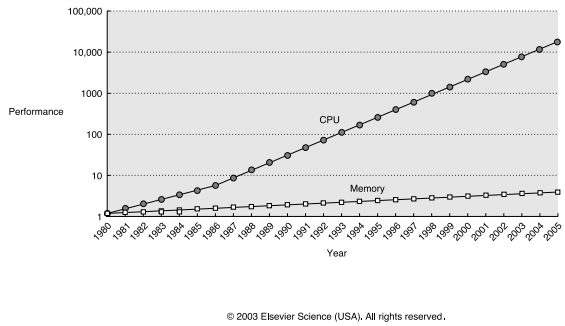

図 1: [3] 第 5 章図 2 からの引用

その結果、CPU とメモリの性能のギャップは年々広がっている。例えば、最近のプロセッサでは、メモリアクセス (キャッシュミス) のペナルティは 200 倍近くあり、メモリアクセスのコストは一定であるという仮定のもとのプログラミングモデルは破綻している。(表 1) 従って、より高性能なソフトウェアを実現にするためにメモリ階層を意識したプログラミングモデルが必要となってきている。

90 年代の研究によれば、SPEC ベンチマークに代表する科学技術アプリケーションにおける CPI (Clock Per Instruction) に比べ、商用ワークロード (OLTP – On Line Transaction Processing) の CPI が大きいことが知られている。その要因の一つがメモリアクセス時のストールである。メモリへのアクセス待ちが CPI を上げるのである。([4])

特にＲＤＢＭＳ等におけるメモリアクセスは、ポインタを利用したランダムなアクセスが多く、科学技術アプリケーションでみられる単純な配列の順アクセスに対する最適化は単純には適用できない。

そこでメモリアクセスに注目した性能向上ツールが必要とされているのだが、従来のタイマ割り込みによる PC(Program Counter) のサンプリング手法でのプロファイリングでは、キャッシュミス等のメモリの動的特性についての詳細情報を得ることができない。そのためプログラマは、おおまかなホットスポット (実行時間を多く消費している場所) の位置を推定することができても、何故そこで実行時間がかかっているかを特定することが困難であった。命令がストールしているのは、データメモリのキャッシュミスなのか？分岐予測の失敗なのか？それとも単に実行にコストがかかる命令を実行中だったのか等について情報が得られない。

2 目的

そこでわれわれは Pentium 4/Intel Xeon におけるハードウェア性能モニタ機能を利用してメモリプロファイリングツールを実装し、RDBMS (PostgreSQL) や Linux Kernel のプロファイリングを実施し、そのツールの有効性を検証した。我々のツールを利用すれば従来のツールでは困難だったメモリイベント (L1/L2 キャッシュミスなど) のホットスポットを容易に発見することができた。

具体的には以下の開発および評価を行った。

1) Linux Kernel およびアプリケーションプログラム（例 OLTP）のメモリアクセスに注目した詳細な動的プロファイルを取得するツールを開発した。Intel Architecture で実装されているハードウェアイベントカウンタなどを利用して CPU クロック数、各種イベント（キャッシュミスなど）を取得することが可能なツールを開発した。
2) 今回開発したメモリプロファイリングツールなどを利用して OLTP ベンチマーク等を実施しメモリアクセスに注目した詳細な動的プロファイルを取得し

ボトルネックを明らかにした。
3) メモリアクセスの動的特性に注目し、各種チューニングを施し評価した。
4) 当該ツールが Linux Kernel および DBMS 等のメモリ性能向上に必要な情報を提供しているかの有効性を確認した。

3 開発内容

Intel 社の 32 ビットマイクロプロセッサ (IA-32 と記す) はモデル固有のハードウェア・パフォーマンス・モニタリング機能を持つ。([5])

われわれは IA-32 のパフォーマンスモニタリング機能を利用して、メモリプロファイリングツールを実装した。

3.1 設計

3.1.1 IA-32 のパフォーマンスモニタニング機能

ここでは簡単に IA-32 のパーマンスモニタリング機能について紹介する。([5])

パフォーマンスカウンタ (PMC) は Pentium から実装され、さまざまなハードウェアイベントの計測を可能としている。計測できるイベントはアーキテクチャモデルによってことなる。最新の Pentium 4 および Intel Xeon プロセッサでは、18 個の 40 ビットのパフォーマンスカウンタを持ち、多くのイベントを同時に計測することができるようになった。(計測できるイベント例：分岐命令数、分岐予測失敗数、バストランザクション数、キャッシュミス数、TLB ミス数、実行命令数等々多数)

タイムスタンプカウンタ (TSC) は、ハードウェアリセット時に 0 から開始し、プロセッサのクロックサイクル毎に増加する 64 ビットのレジスタである。RDTSC 命令によって、カウンタの値を読む。(非特権命令)。ユーザーモードからも読めるので、簡単に実行時のクロック数を計測するのに利用できる。例えば、あるルーチンの実行コストは、入口と出口で TSC を読み、その差が実行コスト (サイクル数) である。(gcc でのコード記述例、下記参照)

コード 1：

```
/* rdtsc 命令の利用例*/
#define rdtscll(val) \
     __asm__ __volatile__("rdtsc" : "=A" (val))

unsigned long long before;
unsigned long long after;

rdtscll(before);
/* 計測する部分*/
rdtscll(after);

diff=after-before; /* 実行クロック数 */
```

パフォーマンスカウンタは Pentium から実装され、Pentium 4/Intel Xeon で様々な拡張がおこなわれた。

3.1.2 Pentium 4/Intel Xeon における精密なイベントサンプリング

最近のプロセッサは、(1) 深いパイプラインを持つ、(2) 投機的な実行をする、(3) スーパースカラで同時に複数命令を実行する、(4) アウトオブオーダー実行する、などの特徴を持つため、サンプリングしたプログラムカウンタ (PC) と実際にイベントを発生させた PC が一致しないという問題があった。([6]、[7])

この問題を解決するために Pentium 4 系プロセッサで精密なイベントサンプリング (PEBS – Precise Event Based Sampling) という機能が実装された。

カーネル空間にデバッグストア (DS – Debug Store) 領域というのを確保できる。パフォマンスカウンタが PEBS 用に設定されている場合、カウンタのオーバフローが発生する都度、プロセッサは汎用レジスタ、EFLAGS レジスタ、インストラクションポインタを DS 領域にある PEBS バッファの PEBS レコードに自動的に格納する。これはマイクロコードによって行なわれる (ハードウェアが自動的に実行する) ので、プロセッサの精密な状態を保存が可能となっている。そしてプロセッサはパフォーマンスカウンタの値をリセットし、カウンタをリスタートする。DS 領域に閾値を設定しておくと、それを越えるレコードが格納された時、PMI 割り込みが発生するので、DS 領域のデータをユーザー空間に退避するようなデバイスドライバを用意しておく。

PMC は 40 ビットなので、2^{40} 回イベントを計測するとオーバーフローする。これを利用して、PMC にあらかじめ 2 の補数をセットしておけば、その回数毎にイベントをサンプリングできるのである。例えば、-100 を PMC にセットすれば、100 回目にオーバフローが発生するので、PEBS の機能 (ハードウェアの機能) によって、イベントが発生したプログラムカウンタ (PC)、各種レジスタの値などを保存される。これをパフォーマンスイベントサンプリングと呼ぶ。

PEBS は At-Retirement イベントのサブセットだけをサポートしている。

従来の割り込みによるサンプリングだと、PMC がオーバフローする度に割り込みが発生し、その割り込み処理ルーチンで各レジスタ値を保存していた。そのため、1) 割り込み処理ルーチンが起動されるまでのレイテンシ、2) 割り込み処理ルーチンそのもののオーバヘッド、などの影響が無視できなかった。そしてイベントを発生させた PC を正確にサンプリングできなかった。([6]、[7])

PEBS によって、それらの問題が解決された。

3.2 実装

ここでは本ツールの実装について記す。

我々は、IA-32 のパフォーマンスモニタリングファシリティを利用して、メモリプロファイリングツールを Linux 上に実装した。

ツールは以下のコンポーネントからなる。

1) メモリプロファイリング用ドライバ (Linux Kernel へのパッチ)
2) ユーティリティ (ebs)
3) ユーザプログラム用 API (Application Programming Interface)

パフォーマンスモニタリングファシリティの機能は Linux ではデフォルトでは利用できないので、それを利用可能にするパッチが必要である。(表 2)

表 2: パフォーマンスファシリティ用パッチ例

名前	開発者名	URL
hardmeter	吉岡弘隆	[12]
perfctr	Mikael Pettersson	[8]
brink_abyss	Brinkley Sprunt	[9]
Rabits	Don Heller	[10]
PAPI	Philip J. Mucchi, et. al	[11]

そこでわれわれは perfctr([8]) をベースに PEBS の機能を追加し、簡単にメモリプロファイリングができるツールを開発した。([12])

http://sourceforge.jp/projects/hardmeter/

3.2.1 ユーティリティ (ebs コマンド)

ユーザーが簡単にメモリプロファイリングをおこなえるようなユーティリティ (ebs コマンド) を開発した。

表 3: ebs の主なファイルとその機能

ファイル名	機能
ebs.c	メインルーチン
hardmeter.c	/proc/pid/perfctr ドライバ
hardmeter.h	include file
p4_template.c	Pentium 4 の定義ファイル
rc.c	resource file (.hardmeterrc) の処理
self.c	

ebs のコマンドシンタックスは下記の通り。

```
Usage: ./ebs (-u | -k) [-o OUTFILE] [-i INTERVAL] \
[-c COUNT] -t TYPE EXE_OR_PID
  option
    -u - sample user-mode events
    -k - sample kernel-mode events
    -o - store sampled data to file
    -i - sampling interval(default: 10000)
    -c - max sampling count(default: 2000)
    -t - event type
```

詳細なコマンドシンタックスおよび利用方法に関しては第 3.4 節を参照されたい。

3.2.2 ユーザープログラム用 API(Application Programming Interface)

ユーザーアプリケーションから簡単に呼べるような高レベル API とより詳細な機能を提供する低レベル API の 2 種類を実装した。

高レベル API ではサンプリングの定義等をユーザーデフォルトディレクトリのリソースファイル (`$HOME/.hardmeterrc`) に定義しておき、アプリケーションプログラムは `hardmeter_start(''resource_name'')` と `hardmeter_stop()` でプロファイリングの開始、停止をおこなう。

リソースファイルの定義例：リソースファイルは、

リソース名 . パラメータ名　パラメータの値

という書式で記述する。リソース名はスペースと '.' 以外の任意の文字を使用できる。resource_name というリソース名でプロファイリングを行う場合は、`$HOME/.hardmeterrc` を下記のように記述する。

```
# sample events in user-mode when non-zero. default 0.
resource_name.user     1
# sample events in kernel-mode when non-zero. default 0.
resource_name.kernel   0
# specify event name.
resource_name.event    l1_cache_miss
# output file
#   %Y  -  year (1970...)
#   %y  -  year (00..99)
#   %m  -  month (01..12)
#   %d  -  day of month (01..31)
#   %H  -  hour (00..23)
#   %M  -  minute (00..59)
#   %S  -  second (00..60)
#   %P  -  process id
resource_name.dumpfile /tmp/hardmeter.%y%m%d-%H%M%S.%P
# max sampling count. default 2000, max 524270 for pebs.
resource_name.count    200
# sampling interval. default 10000.
resource_name.interval 10000
```

プロファイリングの定義は、リソースファイル (`$HOME/.hardmeterrc`) にあるので、プログラムを変更しないでリソースファイルを変更するだけで簡単にプロファイリングができる。

低レベル API の場合、よりきめのこまかい制御が可能になっている。

hardmeter の元となった perfctr では、測定するイベント毎に「どのパフォーマンスモニタリングカウンタを使用するのか、また、それぞれのカウンタの各ビットにどのような値を設定するのか」を明示的に指定する。これらの情報を指定するには Intel のマニュアル ([5]) と首っぴきにどこにどのような値を入れる必要があるかを調べる必要がある。

hardmeter ではユーザの便のため、これらのイベントをテンプレートとしてまとめて、テンプレートの名前を指定するだけで必要な情報を得られるようにした。hardmeter を利用する場合は、使用するテンプレートと若干のパラメータを指定するだけでイベントの測定ができるようにしてある。テンプレートの構造は以下のようになっている。

name にテンプレートの名前、description にその説明がはいる。control, eventmask はテンプレートの実際の設定がはいっている。is_pebs にそのテンプレートが Precise EBS なのか、Non-preciese EBS なのかの区別がはいっている。ユーザプログラム側ではこれらの値の参照のみを行なう。

name のみが定義されていて、他のメンバーが NULL のエントリもあるが、これは ebs コマンドが画面にイベント一覧を出力するときのコメントとしてあつかわれる。

3.3 評価

PEBS (Precise Event Based Sampling) を利用したイベントサンプリングがどのくらい有用な情報を提供するか検証するために、以下のような Intel Xeon マシン (表 4) で評価をおこなった。

表 4: 評価に使用したマシン

評価に使用したマシン	
CPU	Intel Xeon
Clock	2.0GHz
Memory	1GB
L1 cache (Data)	8KB (4 way set associative)
L1 line size	64 byte
Trace cache (Instruction)	12K μ OP
L2 cache (unified)	512KB (8 way set associative)
L2 line size	128 byte (read) or 64 byte (write)

われわれは評価用に次の 2 つのベンチマークを実施した。(表 5)

表 5: 評価に利用したベンチマーク

ベンチマーク	測定方法	参考文献
Linux Kernel Build	make -j	[1]
OLTP TPC-B benchmark	pgbench	[2]

3.3.1 Linux Kernel Build

Linux Kernel 2.4.18 を下記のようにビルドし、その時の L1/L2 キャッシュミスを測定した。`make -j` は、同時に make するジョブを制限なしに並行して実行する。百を越える make が同時に走ることになり、非常に高い負荷を簡単にかけることができる。

```
$ make -j bzImage
```

PEBS の測定結果を分析したところ、Linux Kernel のスケジューラにおいて L1 および L2 キャッシュミスが多発していることを発見し、その原因がコンフリクトミスであることを特定した。

PEBS を利用することによって、容易にメモリトラフィックのボトルネックを発見でき、その原因を特定できた。

詳細は ([1]) で報告したので、参照いただきたい。

3.3.2 OLTP ベンチマーク

PostgreSQL のディストリビューションに標準的に添付されている pgbench を利用して、L1/L2 キャッシュミスを測定した。トランザクションのモデルは TPC-B 型の単純なモデルである。

PEBS の測定結果を分析したところ LWLockRelease()/LWLockAcquire() というルーチンで L1 キャッシュミスが多発していることを発見し、それがスピンロックの実装によることを特定した。スピンロックに対する簡単な改良をほどこし L1 キャッシュミスが 15 %ほど減少していることを確認した。

詳細は ([2]) で報告したので、参照いただきたい。

3.3.3 PEBS の効果

この 2 つのベンチマークテストによって、PEBS を利用することによって下記の効果を確認した。

1) キャッシュミス等ハードウェアイベントのホットスポット (ボトルネック) を精密に特定できた。
2) キャッシュミスの原因究明のヒントを提供した。

3.4 利用方法

ebs は下記のように利用する。

```
Usage: ./ebs (-u | -k) [-o OUTFILE] [-i INTERVAL] [-c COUNT] -t TYPE EXE_OR_PID
  option
    -u - sample user-mode events
    -k - sample kernel-mode events
    -o - store sampled data to file
    -i - sampling interval(default: 10000)
    -c - max sampling count(default: 2000)
    -t - event type
  event type
    (imprecise at-retirement event)
      instr_retired   - instruction retired
      uop_retired     - uops retired
    (precise front-end event)
      memory_loads    - memory loads
      memory_stores   - memory stores
      memory_moves    - memory loads and stores
    (precise execution event)
      packed_sp_retired - packed single-precision uop retired
      packed_dp_retired - packed double-precision uop retired
      scaler_sp_retired - scaler single-precision uop retired
      scaler_dp_retired - scaler double-precision uop retired
      64bit_mmx_retired - 64bit SIMD integer uop retired
      128bit_mmx_retired - 128bit SIMD integer uop retired
      x87_fp_retired  - floating point instruction retired
      x87_simd_memory_moves_retired - x87/SIMD store/moves/load uop retired
    (precise replay event)
      l1_cache_miss   - 1st level cache load miss
      l2_cache_miss   - 2nd level cache load miss
      dtlb_load_miss  - DTLB load miss
      dtlb_stor_miss  - DTLB store miss
      dtlb_all_miss   - DTLB load and store miss
      mob_load_replay_retired - MOB(memory order buffer) causes load replay
      split_load_retired - replayed events at the load port.
```

例えば、

```
./ebs -o pebs -u -i 100 -t dtlb_load_miss ps
```

とすると ps コマンドの DTLB load miss (-t dtlb_load_miss) のユーザーモード (-u) イベントを 100 回 (-i 100) ごとにサンプリングし、それを pebs (-o pebs) というファイルへ出力する。

3.5 データの分析方法

上記のコマンドで下記のようなデータを収集したとする。これをどのように分析するか？

結果：PEBS のレコード例

```
eflags linear_ip     eax      ebx      ecx      edx      esi      edi      ebp      esp
00000246 400074e6 40011c6c 40014dc0 080487ed 40011ceb bfffd8f8 40015438 bfffd724 bfffd2cc
00000286 c012936c c1ab71f0 f7501a00 f73b7c30 00000000 00000009 00000000 f73b7c30 f7391e90
00000206 4000b6e3 bfffd821 40014dc0 08058098 08058098 080488d5 00000000 bfffd914 bfffd82c
00000206 4000db63 4002d000 40014dc0 40015ba0 40015e98 40042bfc 40015e98 bfffd894 bfffd7ec
00000282 c012a53a 40037000 f74d8380 f74d8380 40037000 40037f30 00000000 f7c759b0 f7391ea0
00000202 40008da6 00000000 40014dc0 0000004f 4003e484 000063df 40015eb8 bfffd7b4 bfffd6dc
00000202 40008eff 00000330 40014dc0 08048ab8 00000033 40015438 0804868c bfffd7b4 bfffd6dc
00000206 40008eff 00006b10 40014dc0 00000748 40015ba0 40015ba0 40036d0c bfffd7b4 bfffd6dc
00000246 40008eba 40151848 40014dc0 00000071 40015ba0 40015ba0 40015e14 bfffd7b4 bfffd6dc
00000206 40008e37 40015588 40014dc0 00000000 40015d3c 40032a5c 40015d3c bfffd7b4 bfffd6dc
00000206 4000a833 bfffd91c 40014dc0 00000000 bfffd914 4003334c 40015ec8 bfffd914 bfffd7cc
00000202 40008ff3 0804823c 40014dc0 00000026 40015438 40015438 080485bc bfffd7b4 bfffd6dc
00000202 40008eff 00002ae0 40014dc0 000003e3 40015ba0 0d696910 40032cdc bfffd7b4 bfffd6dc
00000202 40008fac 4004196f 40014dc0 4004196f 40041949 0d696910 4003648c bfffd7b4 bfffd6dc
00000202 40008fac 4004196e 40014dc0 4004196e 4004194c 0d696910 4003815c bfffd7b4 bfffd6dc
00000206 40008e2e 400153a4 40014dc0 00000000 40015d3c 400337fc 40015d3c bfffd7b4 bfffd6dc
00000246 40008eae 40015384 40014dc0 00000000 40015438 40015438 40015e14 bfffd7b4 bfffd6dc
00000297 4000a7f9 00000002 40014dc0 00000000 40015ba0 bfffd914 40015f08 bfffd914 bfffd7cc
00000202 4000a74b 4002d000 40014dc0 4002d000 4014e49c 0012149c 40015e98 bfffd914 bfffd7cc
00000206 40008e2e 400153a4 40014dc0 00000000 40015d3c 4003562c 40015d3c bfffd7b4 bfffd6dc
00000202 40008eff 00000230 40014dc0 08048ab8 00000023 40015438 0804858c bfffd7b4 bfffd6dc
00000246 40008ec3 080487dc 40014dc0 00000000 40015438 40015438 40015e14 bfffd7b4 bfffd6dc
00000206 40008e37 40015588 40014dc0 00000000 40015d3c 4003815c 40015d3c bfffd7b4 bfffd6dc
00000246 40008fac 4004196c 40014dc0 4004196c 4004196c 0d696910 400315bc bfffd7b4 bfffd6dc
00000202 40008eff 000000d0 40014dc0 0000003b 40015438 40015438 0804842c bfffd7b4 bfffd6dc
00000202 40008eff 000003d0 40014dc0 08048ab8 0000003d 40015438 0804872c bfffd7b4 bfffd6dc
00000202 40008f09 00000930 40014dc0 40021778 00000093 40015950 40020eb8 bfffd7b4 bfffd6dc
00000202 40008da6 00000000 40014dc0 0000004f 400409ec 000063df 40015eb8 bfffd7b4 bfffd6dc
00000246 40008ed4 0cd2d046 40014dc0 08048ab8 00000000 40015438 40015e14 bfffd7b4 bfffd6dc
00000206 40008fac 4004196d 40014dc0 4004196d 40041947 0d696910 4003634c bfffd7b4 bfffd6dc
00000202 40008eff 000006e0 40014dc0 40021778 0000006e 40015950 40020c68 bfffd7b4 bfffd6dc
00000246 40008eb1 40029af8 40014dc0 08048ab8 40015950 40015950 40015e14 bfffd7b4 bfffd6dc
00000246 40008e56 00000004 40014dc0 00000000 40015e14 00000000 40015e14 bfffd7b4 bfffd6dc
00000206 40008eff 00000370 40014dc0 00000045 40015180 40015180 40000748 bfffd7b4 bfffd6dc
00000206 40008fac 40041970 40014dc0 40001770 40001742 0d696910 40000a58 bfffd7b4 bfffd6dc
00000202 40008eff 000001d0 40014dc0 40021778 0000001d 40015950 40020758 bfffd7b4 bfffd6dc
00000206 40008eff 00001ad0 40014dc0 00000674 40015ba0 40015ba0 40031ccc bfffd7b4 bfffd6dc
00000206 40008da6 00000000 40014dc0 00000067 40039981 00006887 40015ec8 bfffd7b4 bfffd6dc
00000206 4000907c 40011c6c 40014dc0 4004186a 40011ceb 40015438 40041869 bfffd7b4 bfffd6dc
00000246 4000882c 40014f68 40014dc0 400419ea 40015180 40015180 40015e14 bfffd7b4 bfffd72c
00000206 4000a765 4003284c 40014dc0 4003284c 00000006 0012451c 40015eb8 bfffd914 bfffd7cc
以下略
```

1) ホットスポットの発見。どの場所でキャッシュミスが多いのか？
   PEBS レコードの 2 欄目がイベントが発生した論理アドレス (eip) なので、それをソートして頻度を数えれば、どこの個所でキャッシュミスが多発しているのか発見できる。
2) キャッシュミスの原因の特定
   論理アドレスから、イベントが発生している個所を、オブジェクトを逆アセンブルすることによって、どのアドレスをアクセスしているかがわかる。ソースコードとつきあわせる事によって詳細な検討が可能となる。

3.6 その他特記事項

今回開発したソースコード、各種スクリプトなどはオープンソースとして広く公開している。下記参照。

http://sourceforge.jp/projects/hardmeter/

プロジェクトのホームページには、ソースコードだけでなく各種文書、メーリングリスト、掲示板などがあり、原則として誰でも自由に利用できる。

4 開発成果の特徴

われわれのツールは下記のような特徴を持つ。

1) タイマベースのサンプリングではなく、ハードウェアイベントベースのサンプリングを行う
2) メモリイベントをプロファイルする
3) 精密なイベントベースのサンプリング (PEBS) を行う
4) アプリケーションおよびに Linux Kernel のホットスポットを同定する
5) キャッシュミスの原因究明のヒントを提供する
6) アプリケーションの変更は必要ない
7) オープンソースなので拡張、変更、アプリケーションへの組み込みなど用途に応じたカスタマイズが可能である
8) ベンチマークによって本ツールの有効性を確認した
9) 特別なハードウェアを必要としない

従来のタイマ割り込みを利用したプロファイリングでは、キャッシュミス等のメモリの動的特性について詳細情報を得ることができなかった。そのためプログラマは、おおまかなホットスポット (実行時間を多く消費している場所) の位置を推定することができても、何故そこで実行時間がかかっているかを特定することが困難であった。われわれのツールでは、各種イベントを測定するので、どこでどんなイベントが発生しているのかを同定できるようになった。

最近のプロセッサでは、投機的実行や out of order 実行、深いパイプラインなどにより、イベントを発生させた命令と PC の値が必ずしも一致しないため、上記のプロファイリングだけでは問題を特定することが益々難しくなっている。

われわれは Pentium 4/Intel Xeon で実装された PEBS(Precise Event Based Sampling) という機能を利用することによって、精密なプロファイリングを可能にした。その結果従来のツールでは測定することが不可能だったイベント発生時のレジスタ値の収集などが可能になった。

収集した PEBS レコードを分析することによってアプリケーションおよび Linux Kernel のホットスポット (イベントの多発地帯) を同定できる。PEBS レコードにはイベント発生時の論理アドレス、EFLAGS および各種レジスタ値が格納されているので、論理アドレスでソートし、頻度を数えれば、イベントの多発地帯が発見できる。

イベントが発生した瞬間のレジスタ値が保存されているので、その時にアクセスしているメモリアドレスないしその値がわかる。このアドレスないしレジスタ値がキャッシュミスなどの原因究明のヒントになる。メモリアクセスパターンを分析することによって、たとえばキャッシュコンフリクトによってミスが多発しているなどということが発見できる。

プロファイリングするために、アプリケーションの変更は必要ない。ソースコードないしアプリケーションの再コンパイルおよびビルドなども必要ない。バイナリイメージを実行するだけでよい。

本ツールはオープンソースなので利用者が自由に拡張、変更、アプリケーションへの組み込みなど用途に応じたカスタマイズが可能である。

われわれは簡単なベンチマークを実施して本ツールの有効性を検証した。本ツールを利用することによって、容易にメモリトラフィックのボトルネックを発見し、その原因を特定できた。そして簡単なチューニングをほどこしその効果を確認できた。

本ツールは Pentium 4/Intel Xeon の機能を利用したソフトウェアのみによる実装なので特別なハードウェアモニタ (ハードウェアバスモニタなど) を必要としないため、手軽にプロファイリングが実時間でおこなえる。

5 未踏プログラマとして

未踏ソフトウェア創造事業に採択され約半年間開発してきたわけだが、最後に未踏プログラマとしての感想を述べたい。

わたしは、90 年代中頃、米国に本社のあるデータベースベンダで開発者をしていた。そこには社内で開発された命令数を計測するプロファイリングツールがあった。それを利用して自分が担当するモジュールの命令数を計測してみたところ、簡単にそのモジュールのホットスポットを発見できた。米国に赴任したばかりで、実のところ、そのデータベースエンジンの実装についてはまだ知らないことばかりだったので、担当モジュールの動的特性を知る上でプロファイリングツールが非常に役に立つのだという強い印象を持った。そのツールを利用して、いくつかの改良をほどこしたところ、実行命令数を 20〜30 %削減したことを記憶している。

しかし、そのツールは命令数を計測するだけで、どの命令がどのくらい実行コストがかかるとかの情報は提供してくれなかった。しかも、ユーザーモードしか計測できないので、システムコールのコストやカーネルモードでの実行コストに関する情報は一切提供しなかった。さらに投機的実行での分岐予測ミスなどプロセッサのミクロな挙動についての情報も提供しなかった。

普通のデータベースエンジン開発者にとっては機能開発とバグフィックスがトッププライオリティで、日々の細々したパフォーマンスチューニングは性能上の問題が発覚しないかぎり後回しにされるのが常であった。

ある時性能評価を専門とするエンジニアが Sun の Solaris のエンジニアと共同で、パフォーマンスに関するハックを開発者全員にメールした。それは、例外処理マクロの定義に関するものだった。データベースエンジンでなんらかのシグナルを発行する場合 (例外であったり、単なるメッセージであったり、非同期の処理を起動する場合であったり、様々な場所で使われる) のマクロの処理フローの if 文の条件部を従来は then 側に例外処理、else 側に通常処理という風になっていたのを逆にするだけで、実行時間が 10 数%向上したというものであった。

それは正にハックであった。一つのファイルの一つのマクロの定義を変更するだけで、実行時間を向上させたのである。

SPARC の場合、分岐予測は then 側に行くので、より多く通過する処理を then 側においておくと分岐予測ミスのペナルティを払わなくてすむ。例外処理の場合、例外が発生することはまれであるので、その処理を else 側に記述しておくのが、性能向上に役にたつのである。

普通のデータベース開発者は分岐予測ミスのペナルティを測定するツールを持っていなかったが、Sun のエンジニアは当然なんらかのツールを持っていたのだろう。

ちなみに分岐予測ミスのペナルティの削減は最近ではコンパイラが面倒を見てくれるので、一昔前のように if...then...else で悩まなくてもよい。profile guided optimization として知られる技法は、実行プロファイリングから分岐する確率が高い方へ積極的に分岐するようなコードを生成する。

そんなわけでプロセッサレベルのミクロなチューニングに興味を持ったのだが当時は実際に簡単に利用できるツールがなかったため仕事で利用するまでにはいたらなかった。

一方で、90 年代の中頃から commercial work load の分析というのがさかんにおこなわれてくるようになった。従来の SPEC benchmark などではメモリ利用量が小さかったり、カーネルの利用がほとんどなかったり、あるいは単純なメモリアクセスのため、CPI (Cycle Per Instruction) が極限まで下り (1 以下)、プロセッサの評価の指標として疑問視されはじめてきた。その結果、より大規模なアプリケーションである OLTP (TPC-C) など commercial work load の分析、研究がさかんになってきた。それらは、メモリ利用量が SPEC に代表されるベンチマークより多く、カーネルモードでの実行も多かった。そして実行時間のストールがメモリアクセスによるというのが、それらの研究であきらかになってきた。

わたし自身は 90 年代後半に日本に戻ってきて開発の現場から足をあらいサポート部隊に異動になったのでマイクロアーキテクチャレベルでのチューニングという問題について実装にかかわるということは残念ながらなかった。

2000 年にミラクル・リナックス株式会社の立ち上げに参加したこともあり長らくそのような事を考える事もなかった。

横浜 Linux Users Group (通称 YLUG) では、不定期でカーネル読書会というのを有志で開催しているのだが、その会合で IA-32 の TSC (Time Stamp Counter) の利用方法が話題になった。ユーザーモードで簡単にハードウェアクロック数を計測できるというのを知って、それをきっかけに、いろいろ調べてみると、IA-32 では簡単に各種パフォーマンスイベントを収集できるらしいというのがわかってきた。キャッシュミスや TLB ミスなどメモリトラフィックに関係するイベントもちょっとしたドライバと設定だけで計測できるらしいというのを教えてもらったりした。

簡単なテストプログラムを作ってキャッシュミスのコスト (クロック数) を計測して遊んでみた。

丁度、そのころ未踏ソフトウェア創造事業が平成 14 年度もプロジェクトを公募するということを知り、IA-32 のパフォーマンスモニタリングファシリティを利用したメモリプロファイリングツールを開発するというアイデアで応募したところ幸いにも採択され、今回のプロジェクトにいたっている。

仕事がら Linux Kernel のチューニングに興味があって、カーネルレベルのチューニングをするにはカーネルレベルでのプロファイリングをなんとかしないといけないだろうと思っていた。カーネルハッカーはプロファイリングツールなんかなくとも鼻歌まじりに勘とヒラメキですごいコードを書いて性能向上をするのだろうけど、フツーのプログラマはプロファイリングをとってボトルネックを発見し、ちまちまコードを改良するしかない。

そもそもメモリプロファイリングツールを作りたいという動機は米国にいたころに逆のぼって、あのハックに出あったからだと思う。ちょっとした工夫で性能を劇的に向上させる。

そのためにプロファイリングツールを作りたかった。

そして YLUG のカーネル読書会で IA-32 の機能を知り、おぼろげながら実装のイメージもつかめたのが今回のアイデアである。

いくつかの偶然がかさなりあってプロジェクトが発足した。

しかしプロジェクトを開始するまでまじめに Intel の Software Developer's Manual の第 3 巻 ([5]) を読んだことはなかったし、プロジェクトが実質的に開始した 8 月の中頃に Intel から無料でそのマニュアルが送付されてきて、(なんというタイミングだろう) それをすみからすみまで読むまでは、今回の実装のメインのアイデアである精密なイベントサンプリング (PEBS – Precise Event Based Sampling) というのを知らなかった。

90 年代の研究のサーベイをすればするほど、メモリプロファイリングはネタとしては非常にスジがいいということを確信し、さらに PEBS によって従来では不可能だった精密なメモリプロファイリングができるということを確認したことによってその確信はますます強固になっていった。

従来のプロファイラとは一線を画したのではないかと自負している。

他のマイクロプロセッサで今回のようなツールを開発できるかは不明である。PEBS は Pentium 4/Intel Xeon に依存した機能であるからだ。しかし最近のマイクロプロセッサは各種パフォマンスモニタリングカウンタを持つので (精密でない) イベントベースのサンプリングツールを実装することは可能であろう。

メモリレイテンシとプロセッサの処理スピードのギャップは当面拡大していくことが予想できるので本ツールのようなメモリプロファイリングツールの必要性は益々増えていくと思う。

最後に実装についてふれておく。

インテルの IA-32 のパフォーマンスモニタリングファシリティを利用できるようにしたものとして、表 2 がある。perfctr [8] は P6 および Pentium 4/Intel Xeon に対応している。しかし PEBS には未対応である。abyss [9] は Pentium 4/Intel Xeon には対応しており PEBS にも対応しているが、P6 は対応していない。また Linux Kernel 2.4 系のみに対応している。rabbit [10] は、P6 だけに対応しており、PEBS や Pentium 4 には対応していない。また最近メンテナンスされていないようである。PAPI [11] はマルチプラットフォームなパフォーマンスライブラリである。パフォーマンスモニタリングファシリティの機能については、perfctr [8] を利用しているが、Pentium 4 および PEBS には対応していない。以上のツールは全てオープンソースで、インターネット上で入手できる。PEBS に対応しているのは、現状では abyss [9] だけである。

われわれは perfctr [8] をベースに、PEBS 対応、Linux Kernel 2.4/2.5 対応のドライバおよび設定を容易にする GUI ツールなどを開発した。

開発当初 PEBS に対応していたのは abyss [9] だけであったので予備実験は abyss を利用した。しかし abyss は SMP や HyperThreding に対応していないため開発のベースにすることはなかった。

hardmeter ドライバの実装等は久保氏 (kubo@jiubao.org) にご協力いただいた。YLUG のメンバである。またプロジェクトマネージャの喜連川優東京大学教授には本プロジェクトにおいて様々なご指導をいただいた。ここに記して感謝したい。

6 参加企業および機関

なし。

7 参考文献

[1] 吉岡弘隆, "Intel 系 (IA-32) プロセッサのパフォーマンスモニタリングファシリティを利用したメモリプロファイリングツール", 第 44 回プログラミングシンポジウム、箱根、2003 年

[2] 吉岡弘隆, "OLTP 性能向上を目的としたメモリプロファイリングツール", 第 14 回データ工学ワークショップ、電子情報通信学会、2003 年

[3] Hennessy, J. L., and Patterson, D. A., Computer Architecture: A Quantitative Approach, 3rd Edtion, Morgan Kaufmann Publishers, 2002

[4] Ailamaki, A. et. al, "DBMSs On A Modern Processor: Where Does Time Go?", Proceedings of the 25th VLDB Conference, 1999

[5] Intel, The IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide, Order Number 245472, 2002

[6] Dean, J. et. al, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-Of-Order Processors", Proceedings of Micro-30, December, 1997

[7] Sprunt, B., "Pentium 4 Performance Monitoring Features", IEEE Micro, July-August, 2002,

[8] http://www.csd.uu.se/mikpe/linux/~perfctr/

[9] http://www.eg.bucknell.edu/~bsprunt/

[10] http://www.scl.ameslab.gov/Projects/Rabbit/

[11] http://icl.cs.utk.edu/projects/papi/

[12] http://sourceforge.jp/projects/hardmeter/