メモリプロファイリングツール (hardmeter) の設計と実装

吉岡　弘隆†

† 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

あらまし　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率は CPU に比較して低い。その結果、メモリをアクセスした時のペナルティが年々増加している。特に OS Kernel におけるメモリアクセスはリスト構造の検索あるいはハッシュなどが多数をしめ、科学技術アプリケーションでみられる配列の単純なアクセスに対する最適化は効果がほとんどないことが知られている。本論文は Linux Kernel 性能向上を目的として、パフォーマンスチューニングに必要なメモリプロファイリングツールを開発しその効果を確認した結果を報告する。

キーワード　性能評価, ベンチマーク, キャッシュ, 最適化

The Design and Implementation of A Memory Profiling Tool

Hirotaka YOSHIOKA†

† Miracle Linux Corporation　4-1-30 Akasaka, Minato-ku, Tokyo, 107-0052 Japan
E-mail: †hyoshiok@miraclelinux.com

Abstract　The performance of CPU has been improved more than 50 % per year but the performance improvement of memory in latency is much lower than those of CPU. Therefore the penalty of memory access has been increased every year. Recent works show optimization on scientific workloads is not effective on commercial workloads like OS kernel. We have developed a memory profiling tool to collect performance information for the Linux Kernel and evaluated the effectiveness of the tool.

Key words　Performance Evaluation, Benchmark, Cache, Optimization

1. はじめに

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)

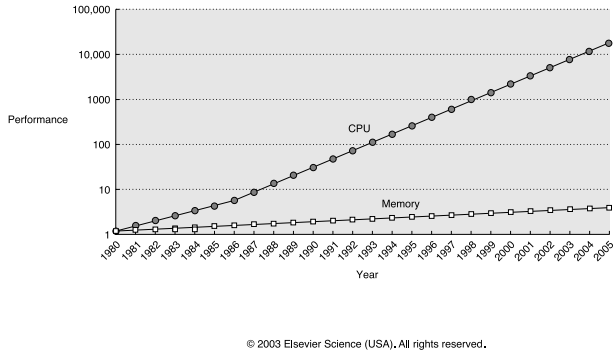

図 1　[1] 第 5 章図 2 からの引用

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。

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

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 OS Kernel 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。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かった。

さらに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。

Pentium 4/Intel Xeon では、上記の問題に対し、ハードウェアによる性能モニタリング機能 (Performance Monitaring Facilities) をそなえ、18 個の性能モニタカウンタ (Performance Monitaring Counters) を持ち、各種メモリアクセスイベント (L1/L2 キャッシュミス/TLB ミス等) を測定できるようになっている。そして PEBS (Precise Event Based Sampling) と呼ばれる機能によって、以前のプロセッサでは不可能だった、より精密なプロセッサの状態のサンプリングが可能になった。

そこでわれわれは Pentium 4/Intel Xeon における上記の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、Linux Kernel においてそのツールの有効性を検証した結果を示す。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。

また、プログラムの小さな変更では、キャッシュミスが多発するプログラムとそうでないプログラムの差を厳密に見ることは、OS のタイマの精度がミリ秒以下の差を測定することが通常困難であるため難しかったが、我々のツールでは、キャッシュミスの回数を測定するので、容易にどちらのアルゴリズムがメモリアクセス特性について優れているか判定できた。そのため、アルゴリズムの抜本的な改良のみならず、OS のタイマの粒度では測定が難しい細かい逐次的改良の積み重ねなどの効果も一つ一つ確認しながら行える。

最後にプロジェクトの今後の方向および課題について述べる。

2. メモリプロファイリングツールの設計と実装

2.1 IA-32 の性能モニタリング機能

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性能モニタリング機能を持つ。([2])

性能カウンタ (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 ビットの性能カウンタを持ち、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。

```
/* 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 */
```

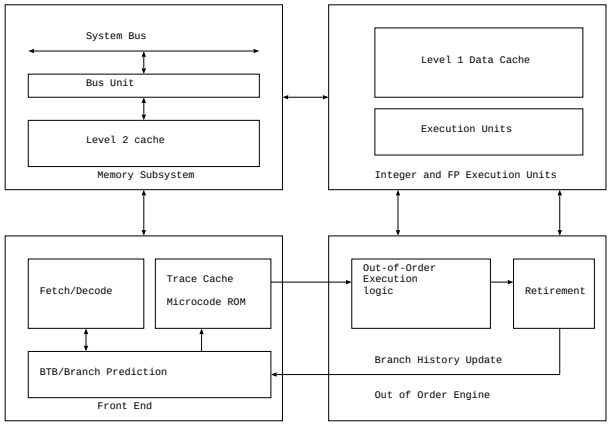

図 2　Pentium 4 ブロックダイアグラム

2.2 Pentium 4/Intel Xeon 系 (NetBurst)

パフォーマンスカウンタは Pentium から実装されたが、以下に示すようにいくつかの限界があった。

- 同時に計測できるイベント数が少ない
- リタイアせずにキャンセルされた命令のイベントも計測する
- イベントサンプリングの精度 (粒度) が荒い

Pentium III(P6 アーキテクチャと呼ばれている) 系プロセッサは投機的な実行をおこなう。したがって分岐予測がはずれた場合、命令をキャンセルするが、キャンセルされた命令が引き起こしたイベントもカウントする。予測がはずれた側の命令が引き起こしたイベント (例えば L2 キャッシュミス) の回数も数えてしまう。この場合、当該イベントが多数発生するのは、分岐予測がはずれたのが主な原因なのか、それとも、L2 キャッシュミスを多発させるようなプログラムなのかは、この情報だけでは判断できない。プログラムを改良する時、分岐予測がはずれないように改良するべきか、それとも L2 キャッシュミスを減らすような改良をするべきか、どちらにプライオリティを置くかなどが判断できないのである。

リタイヤ (コミット) した命令によって発生したイベントと

キャンセルした命令によって発生したイベントを明確に分離できなければならない。

上記の問題を解決するために、Pentium 4/Intel Xeon プロセッサでは下記のような拡張を行っている。

At-Retirement 計測というのは、実際にコミットされた命令にまつわるイベント (non-bogus ないし retire と呼ぶ) と、投機的に実行されコミットされなかった命令 (最終的にキャンセルされた命令) に関するイベント (bogus と呼ぶ) にタグをつけ、命令の retirement 時に区別する機能を指す。(投機的に実行した命令を確定することを retirement するという)

性能カウンタは bogus ないし non-bogus を区別して計測できる。

より粒度の細かい計測を可能にするために、Pentium 4/Intel Xeon プロセッサでは、イベントにタグを付け、コミットした命令だけ (あるいはキャンセルされた命令だけ) を計測する機能を提供した。この機能のことを At-Retirement 計測と呼ぶ。

従来は、実際にコミット (リタイヤ) した命令のみならず、キャンセルした命令によって引きおこされたイベントもカウントしてしまっていた。

イベントサンプリング

性能カウンタ (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。このオーバーフローをトリガーとして、割り込みをかけることができる。割り込みを利用してデータを収集できる。これをイベントサンプリングという。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、割り込みルーチンを利用するか、後述の PEBS(Precise Event Based Sampling) 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。

PEBS (Precise Event Based Sampling) の機能を利用すると、ソフトウェアによる割り込みルーチンを利用しないで精密なサンプリングができる。

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。このため、イベントサンプリングをする場合、ある回数毎の割り込みルーチンが起動され実際のプログラムカウンタ (PC) や各種レジスタの情報を収集する時点では、割り込みルーチンのレイテンシおよび上記のプロセッサの特性により、取得した PC の値は実際のイベントを発生させた PC よりかなり先を行っている。Dean ([5]) らの報告によると、PentiumPro で、取得した PC と実際の PC は 25 命令以上隔たって分布した。Sprunt ([6]) の報告によれば Pentium 4 では 65 命令以上実際の命令と隔たってサンプルされた。プロセッサがより深いパイプラインを持つようになればなるほど、この隔たりは広くなると予想される。また割り込み処理のためのオーバヘッドは無視できない程大きい。

このように Pentium III 以前の割り込みによる方法では、イベントの発生の場所を正確に特定できないため、イベントが発生している時点の正確なコンテキストを入手できない。イベントが発生していることはわかっても、正確なコンテキストがわからないので、性能上何が問題か特定することはこれらの情報だけでは困難である。

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。

性能カウンタが 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 (このデータのことを PEBS レコードと呼ぶ) を DS 領域にある PEBS バッファへ自動的に格納する。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。しかもソフトウェア割り込みによる実装でないため、処理のオーバヘッドはほとんどない。

そしてプロセッサは性能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ことができる。

PEBS は割り込みルーチンを利用しないので、命令キャッシュを汚染しない、データキャッシュをほとんど汚染しない、処理のオーバヘッドが非常に低いという特徴がある。

2.2.1 性能モニタリング機能の使い方

Pentium 4 の性能モニタリング機能は、各種イベントの検出器 (event detectors) とカウンタからなる。性能モニタリング機能を利用して各種ハードウェアイベントを計測する概要は以下のとおりである。

（1） ESCR (Event Selection Control Registor) に計測するイベントとイベントマスクを設定する

（2） 計測するモード (カーネルないしユーザー) を ESCR の OS/USR フラグで設定する

（3） 計測方法は CCCR に設定するので、どの CCCR (Counter Configutstion Control Registor) 選択するか ESCR に設定する

（4） CCCR の compare/complement flags および threshold フィールドを設定する

（5） 必要であれば、CCCR の edge flag を設定する

（6） CCCR の Enable Flag を有効にする

ここで ESCR が各種イベントの検出を行い、CCCR がカウンタの設定を行う。CCCR の Enable Flag が有効になった瞬間から当該イベントの計測が始まり、対応する性能カウンタ (Performance Counter – PMC) にイベント回数が数えられる。

ESCR ないし CCCR の設定は wrmsr 命令で行い、rdpmc 命令によって性能カウンタを読む。

2.3 実　　装

我々は、上記の IA-32 の性能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 Linux 上に実装した。([7])

ツールは以下のコンポーネントからなる。

（1） Linux Kernel へのパッチ (perfctr)

（2） ユーティリティ (ebs)

（３） ユーザプログラム用 API (Application Programming Interface)

性能モニタリングの機能は 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。われわれは perfctr をベースとして利用した。([14])

またイベントの計測を容易にできるようにコマンドユーティリティ(ebs) 作成した。

2.3.1 perfctr への拡張

perfctr 自身は PEBS(Precise Event Based Sampling) 機能がない。従って精密なサンプリングをすることは不可能であった。そこで、われわれは PEBS をサポートするように perfctr の拡張をおこなった。

PEBS 用のデバッグストア領域を準備し、そこに PEBS のレコード (下記) を格納する。PEBS レコードはフラグ、論理命令アドレス、各種レジスタ値などを保持する。

```
struct perfctr_p4_pebs_record {
        unsigned int eflags;
        unsigned int linear_ip;
        unsigned int eax;
        unsigned int ebx;
        unsigned int ecx;
        unsigned int edx;
        unsigned int esi;
        unsigned int edi;
        unsigned int ebp;
        unsigned int esp;
};
```

また PEBS の機能は Pentium 4 以降のプロセッサでしか実装されていないので、Pentium III 以前のプロセッサでは利用できないようなチェックをしている。

上記の PEBS 用デバッグストア領域を用意し、IA32_PEBS_ENABLE というモデル固有レジスタ (MSR – Model Specific Register) に設定をおこなうと、PEBS の機能が有効になり、設定したイベントのサンプリングをハードウェアが自動的におこなうようになる。各種 MSR とフラグの設定方法に関しては Intel のマニュアルを参照していただきたい。([2])

2.3.2 ユーティリティ(ebs)

上記パッチだけではイベントの設定について様々なレジスタ (ESCR や CCCR など) へフラグを 16 進であたえなくてはいけないので非常に繁雑である。そこで、あらかじめ測定すべきイベントに関してはテンプレートを用意して簡単に設定をできるようにした。

またイベントの計測を容易にできるようにコマンドユーティリティ(ebs) 作成した。

計測する対象はユーザモード、カーネルモードどちらも可能である。

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
  options
    -u           - sample user-mode events
    -k           - sample kernel-mode events
    -o OUTFILE   - store sampled data to file
    -i INTERVAL  - sampling interval(default: 10000)
    -c COUNT     - max sampling count(default: 2000)
    -t TYPE      - event type to sample
    -m NAME,NAME... - event masks
  help options
    -h           - show event types
    -h TYPE      - show event masks
```

設定できるイベントは下記の通り。

```
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。

表 2　実験に使用したマシン

実験に使用したマシン	
CPU	Pentium 4
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

3. hardmeter の利用

hardmeter 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 Pentium 4 マシン (表 2) で実験をおこなった。

具体的には、Linux Kernel をビルドして、hardmeter がどのような情報を収集できるかを確認した。具体的には下記のコマンドを実行した。

```
$ /usr/sbin/readprofile -r; \
/usr/src/hardmeter-030603/src/ebs -k -t l1_cache_miss -o l1miss.1 -c 50000 & \
make -j -s bzImage; \
kill -s 2 'ps|grep ebs|grep -v grep|awk '{print $1}''; \
/usr/sbin/readprofile -m /boot/System.map > readprofile.1
```

`make -j -s bzImage;` で同時に複数の make を実行する。システムで可能なかぎりプロセスを fork() して make するため、負荷は非常に高くなる。この時点でのメモリプロファイリング (一次キャッシュミス) を ebs によって計測している。

```
./ebs -k -t l1_cache_miss -o l1miss.1 -c 50000 &
```

は、カーネルモード (`-k`) の一次キャッシュミス (`-t l1_cache_miss`) を最大 50000 レコード (`-c 50000`) まで取得し、それを l1miss.1 というファイルに出力する (`-o l1miss.1`) という事をおこなう。

`readprofile` により kernel デフォルトのプロファイリングを同時に行なう。readprofile はタイマ割り込みにより 10m 秒に一度カーネルのサンプリングを行なう。

3.1 分　　析

hardmeter によって下記のような PEBS レコードを取得した。PEBS レコードは、イベントが発生した時の各レジスタ値 (eflags, linear_ip, eax, ebx, ecx, edx, esi, edi, ebp, esp) を持つ。(PEBS レ

コードは通常のファイルにプレーンテキストとして出力されるので通常の Unix のコマンドで簡単に処理ができる)

```
#eflags  liner_ip   eax      ebx      ecx      edx      esi      edi
00000246 c01124c4 00000002 f73a7800 cb0c0000 e93ef980 e93ef980 00000000
00000202 40103c07 00002f2f 4014bcc4 bfffdc84 bfffffba2 bfffffba2 00002f00
00000206 c0127f11 00000000 00000400 00000010 ece91f8c fe0053c0 bfffd354
00000216 c0148fd7 c1f7d028 f73a2900 00000011 01847743 00000000 f6d7df8c
00000246 c01124c4 00000002 f7c60880 ece90000 e93ef980 e93ef980 00000000
00000206 c0127f11 00000000 00001000 00000400 f6d7df8c fe02d000 40017000
00000283 c01358c5 f79d89a0 f5edcc80 00000010 f79d8900 08099c60 00000010
00000246 c0148fc0 c1fe9810 f6d7df28 00000011 c1f00000 00000000 f6d7df8c
00000246 c01124b4 00000001 f73a7a80 c1be1450 f6d7c000 f39d9080 00000001
...
```

以下略

リニアアドレスは PEBS レコードの 2 桁目 (`awk '{print $2}'`) なので、イベントが発生した時のそれで、ソートし、その回数を以下のように数えた。(表 3)

```
$ grep -v '#' l1miss.1 |\
awk '{print $2}'|\
/usr/src/hardmeter-030603/doc/eip2r -s \
/boot/System.map-2.4.20-hm030603\
 |sort|uniq -c|sort -nr|head
```

表 3　L1 キャッシュミスの頻度と場所 2.4.20 kernel

頻度	アドレス	ルーチン名
2500	c0132298	file_read_actor
1015	c011a843	schedule
959	c01418e8	page_referenced
905	c012e9c6	__constant_memcpy
783	c0141d90	page_remove_rmap
486	c013a34c	scan_active_list
324	c01090a5	ret_from_sys_call
288	c01418fe	page_referenced
261	c0138eb3	lru_cache_add
249	c012ec93	vm_account

リニアアドレスとルーチン名の対応をつけるツール (eip2r) をユーザーの便宜のために hardmeter に同梱している。eip2r のコマンドシンタックスは以下のとおり。

```
Usage: ../doc/eip2r [ -p POSITION ] [ -m PROCESS_MAP ] [ -s SYSTEM_MAP] \
[ -o OUTPUT_FILE ] EIP_FILE
   POSITION    : position of eip in EIP_FILE.
   PROCESS_MAP : output of /proc/<PID>/maps
   SYSTEM_MAP  : System.map of kernel (default: /boot/System.map)
   EIP_FILE    : list of EIP
```

また、当該部分のソースコードはコンパイル時 (gcc) に `-g` オプションを付加しビルドし、objdump コマンドに `-S` オプションを付加することによってえられる。

例えば、schedule() ルーチンの c011a843 付近の逆アセンブルした結果は下記のようになる。

```
objdump -S による出力
                p = list_entry(tmp, struct task_struct, run_list);
c011a840:       8d 4a c4                lea    0xffffffc4(%edx),%ecx
c011a843:       8b 51 28                mov    0x28(%ecx),%edx
/*↑ここでキャッシュミスが多発している*/
```

以下略

上記の逆アセンブル結果より、ECX レジスタで示されるアドレスにアクセスしている時にキャッシュミスを多発していることがわかる。

そこで当該アドレスの PEBS レコードを grep してみる。(横方向はカットしている)

```
#eflags  liner_ip   eax      ebx      ecx      edx      esi      edi
00003002 c011a843 ea29403c ea29403c ea712000 ea71203c ea344000 00000017
00200002 c011a843 eaec803c eaec803c ea740000 ea7403c eb058000 00000016
00200002 c011a843 e9e6803c e9e6803c e9e8e000 e9e8e03c eabb6000 00000017
00200002 c011a843 ea66a03c ea66a03c ea584000 ea58403c eb058000 00000016
00200002 c011a843 e9ff803c e9ff803c f2078000 f207803c eb058000 00000016
00200002 c011a843 e95ac03c e95ac03c e95b2000 e95b203c ea154000 00000015
00200002 c011a843 ea57a03c ea57a03c e9ff8000 e9ff803c ea77c000 00000000
00200002 c011a843 e953803c e953803c e9542000 e954203c ea154000 00000015
00200002 c011a843 f1ba003c f1ba003c e9382000 e938203c ea154000 00000015
00200002 c011a843 e9e5403c e9e5403c e9bce000 e9bce03c ea77c000 00000000
```

ECX の値が xxxxx000 というパターンであることがわかる。(より厳密に言うと下位 13 ビットが等しい)

このキャッシュミスがどのような原因で発生しているか考察してみる。

キャッシュミスが多発しているスケジューラの当該部分は下記のとおりである。

```
list_for_each(tmp, &runqueue_head) {
        p = list_entry(tmp, struct task_struct, run_list);
        /* ↑ ここでキャッシュミスが多発している */
        if (can_schedule(p, this_cpu)) {
                int weight = goodness(p, this_cpu, prev->active_mm);
                if (weight > c)
                        c = weight, next = p;
        }
}
```

ここでは次に実行すべきプロセスを選択しているのだが、実行可能なプロセスは runqueue という単一のキューに繋がれていて、そのエントリをひとつひとつたぐっていくという処理をしている。

`p = list_entry(tmp, struct task_struct, run_list);` によって p に task 構造体のアドレスが代入される。Linux においては task 構造体は 8KB のサイズを持っていて、かつ 8KB のアドレスにアラインされている。

さて、Hennessy および Petterson([1]) にならって、キャッシュミスの原因を次の 3 つに分類する。

- 初期ミス – 最初にアクセスする時に発生するキャッシュミス
- キャパシティミス – キャッシュサイズに比べてワーキングセットが大きい時に発生する
- コンフリクトミス – あるキャッシュラインにアクセスが集中することによって発生する

メモリプロファイリングによって、プログラムのどの個所でキャッシュミス等が発生しているか容易に同定できた。

そして、メモリイベントが発生した時のレジスタの値、メモリアドレスの情報を入手しているので、それをもとに、上記のどれが原因でキャッシュミス等が発生しているかを特定できる。

Pentium 4/Intel Xeon の場合、L1 キャッシュは 8KB、4 ウェイセットアソシアティブ、L2 キャッシュは 512KB、8 ウェイセットアソシアティブなので、それぞれ 4 つないし 8 より多くのアドレスが 2KB(=8KB/4) ないし 64KB(=512KB/8) のモジュロに等しい場合、常に同じキャッシュラインに乗るのでコンフリクトミスが発生する。コンフリクトミスは他のキャッシュラインが空いていても発生するので注意が必要である。

例えばアドレスを 2KB で割った余りが等しい場合、それらは同じキャッシュラインの一つにアサインされる。この時、アドレスの下位 6 bits (64 bytes 分) は、どのキャッシュラインにのるかを決定しない。(キャッシュラインは 64 バイトである) 結局下位 6 bits を無視したアドレスを 2KB で割った余りが等しい場合、それらは同じキャッシュラインの一つにのる。すなわち下記の A の部分が等しいアドレスの場合、同じキャッシュラインの一つにアサインされる。(x は don't care)

```
xxxx xxxx xxxx xxxx xxxx xAAA AAxx xxxx
```

その時、4 ウェイセットアソシアティブキャッシュには同時に

4つのキャッシュラインしか保持できないので、それ以上のデータをキャッシュしようとすると、同じキャッシュラインのどれかと交換しなければならなくなる。すなわちアドレスの bit 位置 10 から bit 位置 6 までが等しい場合にコンフリクトミスが発生する可能性が高い。他のキャッシュラインが空いていてもアクセスするアドレスによって、このコンフリクトが発生する。

Linux Kernel 2.4 系のスケジューラの場合、上記のように 8KB にアラインされている task 構造体を順にアクセスしているので runqueue に繋がれたプロセスが 4 つより多い場合、常にキャッシュコンフリクトを発生させる。ECX の値は下位 13 ビットが等しいので、キャッシュコンフリクトが発生していることが確認できる。

例えばキャッシュコンフリクトの場合、カラーリングとして知られる手法によって、コンフリクトを減らすことができる。Linux Kernel 2.4 の例でスケジューラがコンフリクトミスを多発していることを発見した山村ら ([19]) は、キャッシュのカラーリング手法を利用してスケジューラの L2 キャッシュミス率 (= (L2 キャッシュミス) ／ (L2 キャッシュアクセス)) を 85 %〜90 %から 3 %〜14 %ほどに削減した。

Linux Kernel 2.6 系の場合、スケジューラのアルゴリズムが抜本的に改善され、O(1) スケジューラとして知られるものになった。それは、ひとつの runqueue を持つのではなく、プライオリティ毎にキューを持つことによって、次に実行すべきプロセスの選択のコストを O(n) から O(1) にした。

そこで、O(1) スケジューラが、キャッシュミスにどのような影響を与えるか評価するために、2.4.20 に O(1) スケジューラのパッチを適用し、同様のベンチマークを実施した。

スケジュラーでのキャッシュミスが発生していないことが確認できた。アルゴリズムを変更することで、キャッシュミスを劇的に減少させることができた。(表 4)

表 4　L1 キャッシュミスの頻度と場所 2.4.20 O(1) kernel

頻度	アドレス	ルーチン名
2463	c0136438	file_read_actor
886	c0132af6	__constant_memcpy
660	c0145ec0	page_remove_rmap
349	c014a7e7	invalidate_bdev
266	c0145d90	page_add_rmap
260	c0140354	rmqueue
258	c010958d	ret_from_sys_call
237	c013230f	do_anonymous_page
227	c013d053	lru_cache_add
217	c0133b63	find_vma

キャパシティミスか、初期ミスかは、ソースコードを確認することによって判断できる。同じアドレスがよくキャッシュミスを発生している場合はキャパシティミスをうたがってみる。

キャパシティミスに関しては、ブロッキングとして知られている方法によって、ワーキングセットを減らせる場合がある。

初期ミスに関してはプリフェッチをおこなうことによってレイテンシを下げることができる場合がある。あるいはデータ構造を工夫して、データサイズを減らし、一回のアクセスでいくつかの関連するデータを一度にキャッシュラインにのせることなどで対処できる。

いづれにせよメモリプロファイリングによって選られた情報とソースコードを分析することによってキャッシュミスを削減することが可能である。

4.　hardmeter の評価

4.1　実行オーバヘッド

カーネルビルド時のメモリプロファイリングを ebs コマンドで行なった時のオーバヘッドを time コマンドによって計測した。メモリプロファイリングを取得した場合と、しない場合についてそれぞれ 3 回計測し平均を取った。(表 5)

ebs コマンドのオーバヘッドはいずれも 1 %未満であった。

表 5　ebs コマンドの実行オーバヘッド

	ebs あり	ebs なし	オーバヘッド
real	4m05.043s	4m04.343s	0.29 %
user	3m46.703s	3m46.350s	0.16 %
sys	0m12.883s	0m12.760s	0.96 %

PEBS レコードの取得はハードウェアにより実装されているのでオーバヘッドはほとんどないと考えられる。仮にサンプリングを割り込みにより実装していたとすると、キャッシュミスが発生するたびに割り込みハンドラが起動するので、そのオーバヘッドは大きいと考えられる。また割り込みハンドラは命令キャッシュを汚染するので、計測しているプログラムの挙動 (命令キャッシュミス等) にも影響をあたえる。PEBS 利用の場合はそのような影響はない。

4.2　メモリアクセスのプロファイリングについて

hardmeter は PEBS の機能を利用しているので、どのメモリにアクセスしているか精密に測定できる。従来のプロファイリングツールでは精密にレジスタ値を測定できなかったので、どのメモリをアクセスしているか実時間で計測することは不可能であった。

メモリアクセスの詳細なプロファイリングはソフトウェアによる実装ではシミュレーションで取得するしかなかったが、実行オーバハッドが非常に大きいため、実時間でのデータ収集は不可能であった。実時間での収集は特別なハードウェアが必要であったが、今回の実装では特に必要ではない。

従来のプロファイリングツールでは、メモリアクセスのプロファイリングができなかったので、キャッシュコンフリクトを発見したりすることはできなかったが hardmeter では前述のように容易に発見できる。

4.3　その他の特長

hardmeter はカーネルモードおよびユーザーモードどちらのプロファイリングも可能である。

測定すべきプログラムのコンパイルやリンクは必要でない。またソースコードがなくても測定できる。

5.　今後の方向と課題

Pentium 4/Intel Xeon プロセッサが持つ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について報告した。プロジェクトの今後の方向、課題などを述べる。

5.1 ツールの改良

ツールの改良の方向として、コマンドモードのインターフェースから GUI への拡張、プロファイリング結果のビジュアル化 (リアルタイムなグラフ化等) などを行ない、一般ユーザーの利用の敷居を下げる。

また hardmeter をインストールするために、カーネルの再構築をしないとならないが、カーネルを再構築しなくてもインストールできるように改良する。(カーネルモジュールとして実装する)

5.2 ユーザーグループの構築

ツールの完成度 (実用性、利便性、適応可能性など) を高めるためにユーザーグループを組織し利用ノウハウを蓄積し共有する必要がある。実際に利用している人々の声をフィードバックしツールの改良にやくだてる必要がある。

メモリプロファイリングという分野は他に例を見ないため、十分な利用ノウハウをわれわれは持ちあわせていない。プロファイリングデータを効果的に分析し、それをチューニングに生かすには、ツールのノウハウだけでなくアプリケーションドメインの深いノウハウが必要で、それは実際のユーザーが持つ。

5.3 ベンチマークの実施とチューニング

本ツールの有効性を示すために、われわれはベンチマークを行ったが、今回行ったアプリケーションドメイン以外 (Linux Kernel/OLTP) でもベンチマークを実施し、ツールの有効性を示す必要があるだろう。

さらに、原因の特定などを行ない実験的なチューニングを実施し、再度ベンチマークを行ない、チューニング前後の差異を容易に認識できることを確認する。

われわれはメモリプロファイリングがパフォーマンス分析およびチューニングツールとして非常に有効だと確信するが、エンドユーザにとって判りやすい事例を収集し、有効性をわかりやすく表現することが重要だと考える。

5.4 キャッシュコンシャスなアルゴリズムの適用

今回はメモリプロファイリングツールの評価のみを行なったが、いろいろなキャッシュコンシャスなアルゴリズムを実装し、評価する必要がある。

ミクロな改良として、投機的なプリコンピュテーション、深いデータプリフェッチ、メモリブロッキング、カラーリングおよび新規のアイデアなどを実装し、その効果を各種ベンチマークのメモリンプロファイリングをとることによって定量的に確認する。

プログラムの機械的な改良、あるいは逐次的な改良ではコンスタントオーダーの改良しか期待できないが、アルゴリズムの改良は計算量のオーダーが変更する可能性がある。

また従来のアルゴリズムの評価基準は主に計算量であった。しかし今まで議論したように、キャッシュにヒットするかしないかで最大数百サイクルの差がでてくる。計算量だけではなくて、メモリ階層を考慮したアルゴリズムの評価が重要になってくる。例えば、従来よく知られているアルゴリズムなどをキャッシュを意識した実装とそうでない実装などの比較、評価が必要となってくる。

5.5 チューニングのパターン化

PEBS によってイベントが発生した時の詳細な情報が入手可能となった。それによってキャッシュミス時の要因もある程度特定できるようになった。そこで、各要因に関しての対処方法などをまとめれば一つのチューニング方法論になる。

High Performance Computing(HPC) の分野で知られている高速化のノウハウをパターン化し、メモリプロファイリングツールで発見される現象と関連付ける。

例えばメモリアドレスが、2KB のモジュロで等しければコンフリクトミスが発生するので、その場合は、既知の方式の何々でチューニングするとかいうノウハウをまとめることが必要である。

しかし Linux Kernel は、HPC のような配列に対するループによるアクセスがほとんどなくて、線型リストをポインタで手繰るような処理が多いので、単純には適用できないものが多いと推定される。

6. ま と め

CPU とメモリ処理速度の向上率の差から、今後益々メモリ構成を意識したプログラミングが重要になってくる。しかしながら従来はメモリアクセスのコストの差を簡単に指摘するツールがなかったためメモリアクセスに優しいプログラムを書くことが難しかった。

hardmeter を利用すれば、簡単にユーザーにメモリ関連の動的特性に関する情報を提供でき、性能上のボトルネックを発見できる。その情報をもとにチューニングすることができる。

7. 謝　　辞

本プロジェクトは平成 14 年度未踏ソフトウェア創造事業 (プロジェクトマネージャー喜連川優東京大学教授)「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」として支援を受けた。hardmeter の着想のヒントは横浜 Linux Users Group (YLUG) のメーリングリストでの議論および不定期に開催しているカーネル読書会で得た。実装に関しては、YLUG の久保氏の協力を得た。ここに記して感謝したい。

また、今回開発したスクリプト等は下記の URL で公開している。またメーリングリストなどもあるのでご活用いただければ幸いである。http://sourceforge.jp/projects/hardmeter/

文　　献

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

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

[3] Hinton, G. et. al, "The Microarchitecture of the Pentium 4 Processor", Intel Technology Journal, Q1 2001 Issue, February 2001

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

[5] Dean, J. et. al, "ProfileMe: Hardware Support for Instruction-Level

Profiling on Out-Of-Order Processors”, Proceedings of Micro-30, December, 1997

[6] Sprunt, B., “Pentium 4 Performance Monitoring Features”, IEEE Micro, July-August, 2002,

[7] 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

[8] 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”, データ工学ワークショップ、電子情報通信学会、2003 年

[9] 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”、平成 14 年度ソフトウェア創造事業、成果報告書、2003 年

[10] 吉岡弘隆、日経ソフトウエア、2003 年、下記にアクセスするには無料登録が必要
未踏ソフトウエア奮闘記 オジサン・プログラマの熱い挑戦
http://itpro.nikkeibp.co.jp/members/NSW/ITARTICLE/20030619/1/
http://itpro.nikkeibp.co.jp/members/NSW/ITARTICLE/20030619/1/
http://itpro.nikkeibp.co.jp/members/NSW/ITARTICLE/20030709/1/
http://itpro.nikkeibp.co.jp/members/NSW/ITARTICLE/20030709/2/

[11] 吉岡弘隆、Pentium4/Intel Xeon における性能モニタ機能を利用した　メモリプロファイリングツールを開発する――基礎知識編インターフェース、2003 年 8 月号

[12] 吉岡弘隆、Pentium4/Intel Xeon における性能モニタ機能を利用した　メモリプロファイリングツールを開発する――実践編インターフェース、2003 年 10 月号

[13] 吉岡弘隆、メモリプロファイリングツール hardmeter について、日経 Linux、2003 年 10 月号

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

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

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

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

[18] Ingo Molnar, ultra-scalable O(1) SMP and UP scheduler
http://www.uwsg.iu.edu/hypermail/linux/kernel/0201.0/0810.html

[19] 山村周史他, “エンタープライズ向けプロセススケジューラの評価および改良”, Linux Conference 2002 年

[20] Intel, Intel Pentium 4 and Intel Xeon Processor Optimization Reference Manual, Order Number 248966, 2002

[21] Sears, C. B., “The Elements of Cache Programming Style”, Proceedings of the 4th Annual Linux Showcase and Conference, 2000

[22] Cantin, J. F., et al, “ Cache Performance for SPEC CPU2000 Benchmarks”, http://www.cs.wisc.edu/multifacet/misc/spec2000cache-data/

[23] Gee, J. D., et al, “Cache Performance of the SPEC92 Benchmark Suite”, IEEE Micro (August) 17-27, 1993